Section 25
Flight Database Function

The *Flight Database Processes (FDPs)* perform the following functions:

- Maintain an entry for each flight (active, proposed, or completed within the past 12 hours) in the National Airspace System (NAS)
- Update the flight records when any new data arrives about a flight
- Distribute updates about each aircraft's current situation to other functions within the ETMS

Processing Overview

Figure 25-1 illustrates the following major tasks the *FDPs* perform:

(1) Match any flight update message to the correct entry in the database, or create a new entry if no match exists.

(2) If the message contains field 10 (route) information, update the list of route events in the database.

(3) If the message contains flight position data, record this.

(4) Use all available information (current position, route, high altitude winds, aircraft dynamics, estimated ground time) to predict the future behavior of the flight (where it will be at any specific time).

(5) If the message modified the current status of the flight, or changed our predictions about the future behavior of the flight, distribute information about the changes to specific places, including

  (a) Any information about flight updates or route changes is forwarded to the *FTM* processes.

  (b) Any information about changes involving route events (which fixes or sectors a flight will fly over/through) or route event times is forwarded to the *TDB* processes.

  (c) Any updates are forwarded to a slave *FDB* process if one exists.

(6) Periodically flush data on old flights (those that landed more than 12 hours ago) to archival processes.

(7) Provide information about flights in the database to other processes on request.

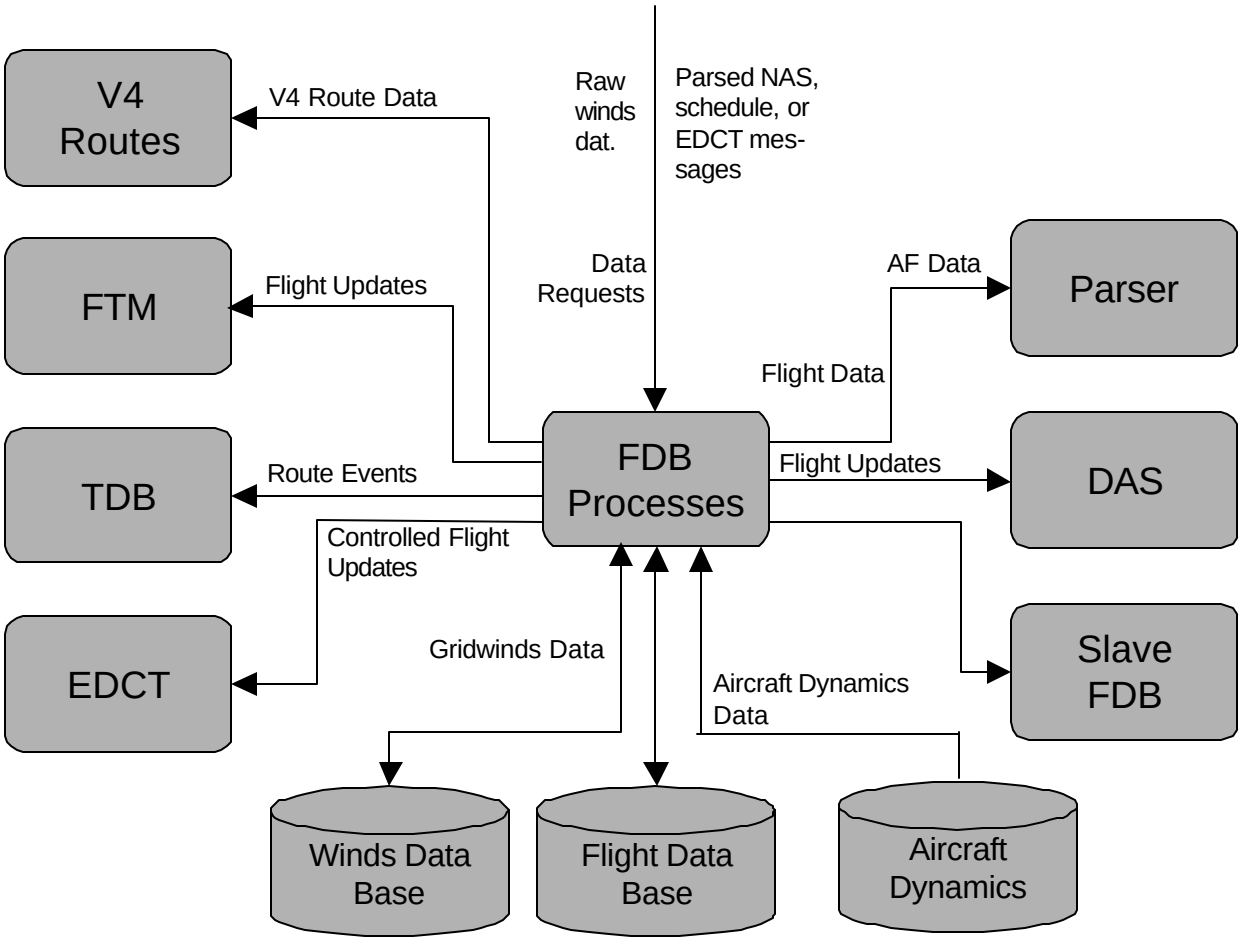

Figure 25-1. Simplified Data Flow involving the Flight Database Processes

The following pages describe several significant design issues involved in the overall conception of the *FDPs.*

Design Issue 1: Unique Flight Indices/Master–Slave

One of the ETMS system goals is to provide a unique index to identify flights across strings. Since some sites are normally receiving data from A string, and others from B string, having the same flight index on both strings allows sites to switch their source of data at will. The method chosen to implement unique flight indices is generalized as

- One of the strings is running as the Master string. The Master is responsible for processing all incoming flight messages, deciding which entry in the database is to be updated, and preparing an off-line picture of what the database entry should look like after the message data has been incorporated. Once the above steps have been completed, the Master plugs the updated picture into its database, and sends a copy of the picture to the Slave FDB.

- The other string is running as the Slave string. The only messages it usually receives are the updated database entry pictures sent over by the Master string, and all it has to do is plug them into the correct slot in its database.

Following these two steps ensures that the database entries for any particular flight reside in the same database slot on the different strings.

Design Issue 2: Database Recoveries

Another of the goals of ETMS is that databases will be recoverable in case of catastrophic failure of one database. To implement this, the following types of recovery/fill-in actions are enabled:

(1) A string's flight database can be recovered from another string if those two strings had been working in a Master/Slave relationship.

(2) A site's FTM database can be recovered from a hubsite *FDP*.

(3) Holes in a site's FTM database can be filled from a hubsite *FTP* to avoid showing partial data in flight lists.

Design Issue 3: Multiple Processes Make Up the FDPs

In order to ensure efficient processing of data, the multiple functions of the *FDPs* have been split into several processes along these general lines of responsibility:

(1) Flight update (NAS) message processing, flight modelling, and database maintenance are handled by the *fdb_manager* process. This process also prepares messages to be dispatched to other ETMS functions, but it relies on various other processes to handle the actual communication duties.

(2) Communication processes that take data prepared by *fdb_manager* and distribute it to other functions:

   (a) *FDB_receiver* handles buffering messages coming into the *fdb_manager* process.

   (b) *tdb_relay* stores and forwards data about route events for each flight to the *Traffic Demands Processor.*

   (c) *edct_relay* stores and forwards data concerning flights that have ground delays associated with them to the *EDCT process.*

   (d) *das_relay* stores and forwards data about flights that are being flushed from the database to an archival process.

   (e) *cross_string_relay* stores and forwards data to a slave *FDB_manager* process.

   (f) *feedback_relay* stores and forwards messages concerning certain AF messages back to the *parser* process.

(g) *route_relay* prepares and ships version 4.2 route messages to *nas.dist* to support the ATA feed.

(h) *fdb_dist* forwards flight transactions to any *FTM* processes that are registered for services.

(3) Requests for specific information about flights currently in the database are handled by two processes: *fdb_router* and *fdb_data_server*.

(4) The *FDB_recovery* process is invoked whenever a cross-string *FDB_manager* recovery is performed.

(5) The *FTM_recovery* process is invoked whenever an *FTM* process has requested a database recovery.

The data flow among these processes is depicted in Figure 25-2.

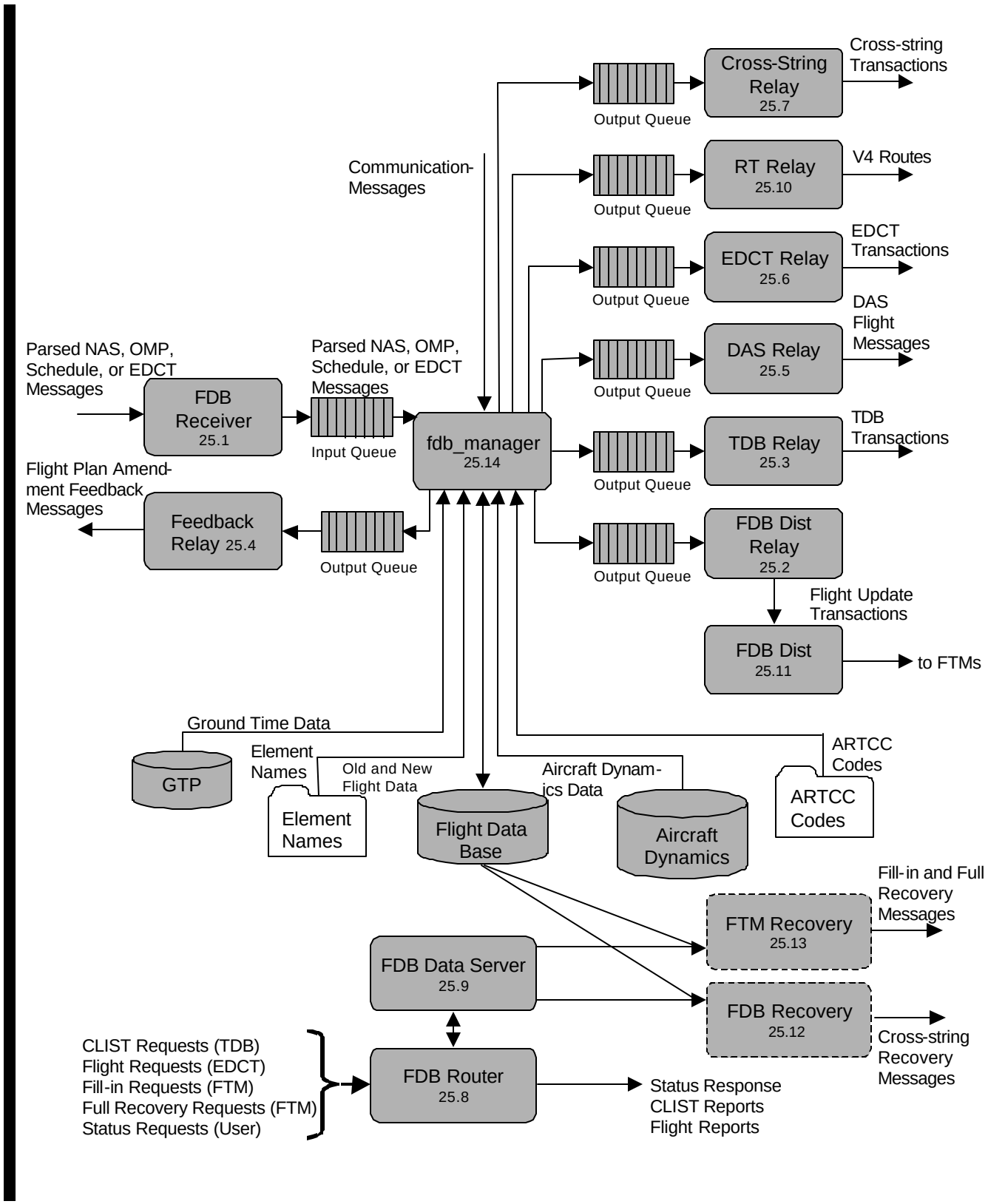

Figure 25-2. Data Flow of the Flight Database Processor

25.1 The FDB Receiver Process

Purpose

The purpose of the *FDB Receiver* is to enqueue the flight state data and route data from parsed NAS, OMP, flight schedule, and EDCT messages for retrieval by the *fdb_manager* process. The *FDB Receiver* buffers the messages in a manner that isolates the *fdb_manager* process from any I/O processing, thus allowing the *fdb_manager* process to process data from the *Parser* asynchronously and ensuring that all messages are processed in the order in which they are received.

Execution Control

The *fdb_manager* process starts the *FDB Receiver* as a child process. This data driven process runs continuously; if it fails, the *fdb_manager* parent process restarts it using the old queue so that a minimum amount of data are lost. Non-fatal errors cause an error message to appear in the receiver window of the ETMS operator's node.

Input

The *FDB Receiver* input consists of flight state data and route data which the *Parser* has extracted from NAS, flight schedule data, and EDCT and OMP messages.

Output

The *FDB Receiver* output is identical to its input.

Processing

The *FDB Receiver*'s processing varies depending on the instance of the *FDP*. In one case, the *FDB Receiver* creates the socket to which the *Parser* and *EDCT Server* connect, and it allocates memory for use as a queue. As messages arrive at the socket, the *FDB Receiver* puts them in the queue for use by the *fdb_manager* process when it's ready. See Section 23.2 for a more detailed description of receiver processes.

Error Conditions and Handling

In general, if the files necessary for creating the queues are locked, *FDB Receiver* unlocks them before creating or opening them.

25.2 The FDB Dist Relay Process

Purpose

The *FDB Dist Relay* process buffers and relays route information generated by the *fdb_manager* process to the *FDB Distributor*. The *FDB Dist Relay* process does not change the data; it merely ensures that all data destined for the *FDB Distributor* make it there.

Execution Control

The *fdb_manager* process starts the *FDB Dist Relay*, a continuously running, data-driven process. If the *FDB Dist Relay* fails, the *fdb_manager* parent process restarts it using the old queue so that minimum data loss occurs. See Section 33 for more information on restarting processes.

Input

The *FDB Dist Relay* receives packets of route information created by the *fdb_manager* process.

Output

The *FDB Dist Relay* output is identical to its input.

Processing

The *FDB Dist Relay* opens a channel to the socket controlled by the *FDB Distributor*. It then continuously dequeues messages and writes them to the channel. See Section 23.2 for a complete description of relay processes.

Error Conditions and Handling

If the *FDB Dist Relay* loses contact with the *FDB Distributor* socket, the *FDB Dist Relay* will attempt to reconnect to the socket. Non-fatal errors cause an error message to appear in the FDB Dist Relay window.

25.3 The TDB Relay Process

Purpose

The *TDB Relay* process buffers and relays update transactions generated by the *fdb_manager* process to the *Traffic Demands Database Processor*. The *TDB Relay* does not change the data; it ensures that all data headed for the *Traffic Demands Database Processor* reach their destination.

Execution Control

The *fdb_manager* process starts the *TDB Relay*, a continuous, data-driven process. If the *TDB Relay* fails, the *fdb_manager* parent process restarts it using the old queue so that a minimum number of data are lost. See Section 33 for more information on restarting processes.

Input

The *TDB Relay* receives the transactions that are used by the *Traffic Demands Database Processor* to update the TDB.

Output

The *TDB Relay* output is identical to its input.

Processing

The *TDB Relay* opens a channel to the socket controlled by the *Traffic Demands Database Processor*. It then continuously dequeues messages and writes them to the channel. Non-fatal errors cause an error message to appear in the TDB Relay window.

Error Conditions and Handling

If the *TDB Relay* loses contact with the *TDB*'s receiver socket, the *TDB Relay* will attempt to reconnect to the socket.

25.4 The Feedback Relay Process

Purpose

The *Feedback Relay* process buffers and relays feedback (FA) messages generated by the *fdb_manager* process to the *Parser* function. The *Feedback Relay* does not change the data; it ensures that all data headed for the *Parser* function reach their destination.

Execution Control

The *fdb_manager* process starts the *Feedback Relay*, a continuous, data-driven process. If the *Feedback Relay* fails, the *fdb_manager* parent process restarts it using the old queue so that a minimum number of data are lost. See Section 23.2 for more information on restarting processes.

Input

The *Feedback Relay* receives transactions generated by the FDB and contains information merged from AF NAS messages and flight database records.

Output

The *Feedback Relay* output is identical to its input.

Processing

The *Feedback Relay* opens a channel to the socket controlled by the *Parser* function. It then continuously dequeues messages and writes them to the channel. See Section 23.2 for a complete description of relay processes.

Error Conditions and Handling

If the *Feedback Relay* loses contact with the *Parser's* feedback socket, the *Feedback Relay* will attempt to reconnect to the socket. Non-fatal errors cause an error message to appear in the Feedback Relay window.

25.5 The DAS Relay Process

Purpose

The *DAS Relay* process buffers and relays the flight information generated by the *fdb_manager* process to the *Ground Time Prediction System*. The *DAS Relay* does not change the data; it ensures that all data headed for the *Ground Time Prediction System* reach their destination.

Execution Control

The *fdb_manager* process starts the *DAS Relay*, a continuous, data-driven process. If the *DAS Relay* fails, the *fdb_manager* parent process restarts it using the old queue so that a minimum number of data are lost. See Section 33 for more information on restarting processes.

Input

The *DAS Relay* receives the packets of flight information created by the *fdb_manager* process.

Output

The *DAS Relay* output is identical to its input.

Processing

The *DAS Relay* opens a channel to the socket controlled by the *Ground Time Prediction System*. It then continuously dequeues messages and writes them to the channel. See Section 23.2 for a complete description of relay processes.

Error Conditions and Handling

Non-fatal errors cause an error message to appear in the main process window of the ETMS operator's node. When a fatal error occurs, the *fdb_manager* process should restart the *DAS Relay*. If the *DAS Relay* loses contact with the *Ground Time Prediction System's* socket, the *DAS Relay* will attempt to reconnect to the socket.

25.6 The EDCT Relay Process

Purpose

The *EDCT Relay* process buffers and relays edct transactions generated by the *fdb_manager* process to the *EDCT Receiver*. The *EDCT Relay* does not change the data; it ensures that all data headed for the *EDCT Server* reach their destination.

Execution Control

The *fdb_manager* process starts the *EDCT Relay*, a continuous, data-driven process. If the *EDCT Relay* fails, the *fdb_manager* parent process restarts it using the old queue so that a minimum number of data are lost. See Section 33 for more information on restarting processes.

Input

The *EDCT Relay* receives the packets of flight state data and control time information created by the *fdb_manager* process.

Output

The *EDCT Relay* output is identical to its input.

Processing

The *EDCT Relay* opens a channel to the socket controlled by the *EDCT Receiver*. It then continuously dequeues messages and writes them to the channel. See Section 23.2 for a complete description of relay processes.

Error Conditions and Handling

Non-fatal errors cause an error message to appear in the main process window of the ETMS operator's node. When a fatal error occurs, the *fdb_manager* process should restart the *EDCT Relay*. If the *EDCT Relay* loses contact with the *EDCT Receiver's* socket, the *EDCT Relay* will attempt to reconnect to the socket.

25.7 The Cross–String Relay Process

Purpose

The *Cross-String Relay* process buffers and relays cross-string transactions (e.g., updated FDB records and event lists) generated by the Master *fdb_manager* process. It relays these transactions to the *FDB Receiver* on the slave string. The *Cross-String Relay* does not change the data; it ensures that all data headed for the the slave FDB string reach their destination.

Execution Control

The *fdb_manager* process starts the *Cross-String Relay*, a continuous, data-driven process. If the *Cross-String Relay* fails, the *fdb_manager* parent process restarts it using the old queue so that a minimum number of data are lost. See Section 33 for more information on restarting processes.

Input

The *Cross-String Relay* receives buffered cross-string transactions created by the master *fdb_manager* processes.

Output

The *Cross-String Relay* output is identical to its input.

Processing

The *Cross-String Relay* opens a channel to the socket controlled by the slave *FDB Receiver*. It then continuously dequeues messages and writes them to the channel. See Section 23.2 for a complete description of relay processes.

Error Conditions and Handling

Non-fatal errors cause an error message to appear in the main process window of the ETMS operator's node. When a fatal error occurs, the *fdb_manager* process should restart the *Cross-String Relay*. If the *Cross-String Relay* loses contact with the slave *FDB Receiver's* socket, the *Cross-String Relay* will attempt to reconnect to the socket.

25.8 The FDB Router

Purpose

The *FDB Router (FDBR)* process responds to *center list (CLIST)* requests from the TDB, Flight Requests from the EDCT, *recovery* requests from the FTM, *reconfigure* requests, and *status* requests. The *FDBR* works in coordination with the *FDB Data Server*s to provide a mechanism for accessing data from the Flight Database without burdening the *fdb_manager*.

Execution Control

Nodescan starts the *FDBR* process. The *FDBR* creates a socket and connects to the node switch via network addressing. It waits for the *FDB Data Server* to connect to the socket and then advances to its primary processing loop waiting for FDB data requests. On receipt of a valid data request, it formats a message and sends it to the *FDB Data Server* process via the socket.

Input

All inputs to the *FDBR* are in the form of FDB data requests. Valid inputs to the *FDBR* are center list (CLIST) requests from the *TDB*, Flight (FDATA) requests from the EDCT, recovery requests from the *FTM*, reconfigure requests, and status requests.

Output

The *FDBR* outputs socket messages containing FDB data requests to the *FDB Data Server* and Clist, Flight, and Status responses to the specified destination address.

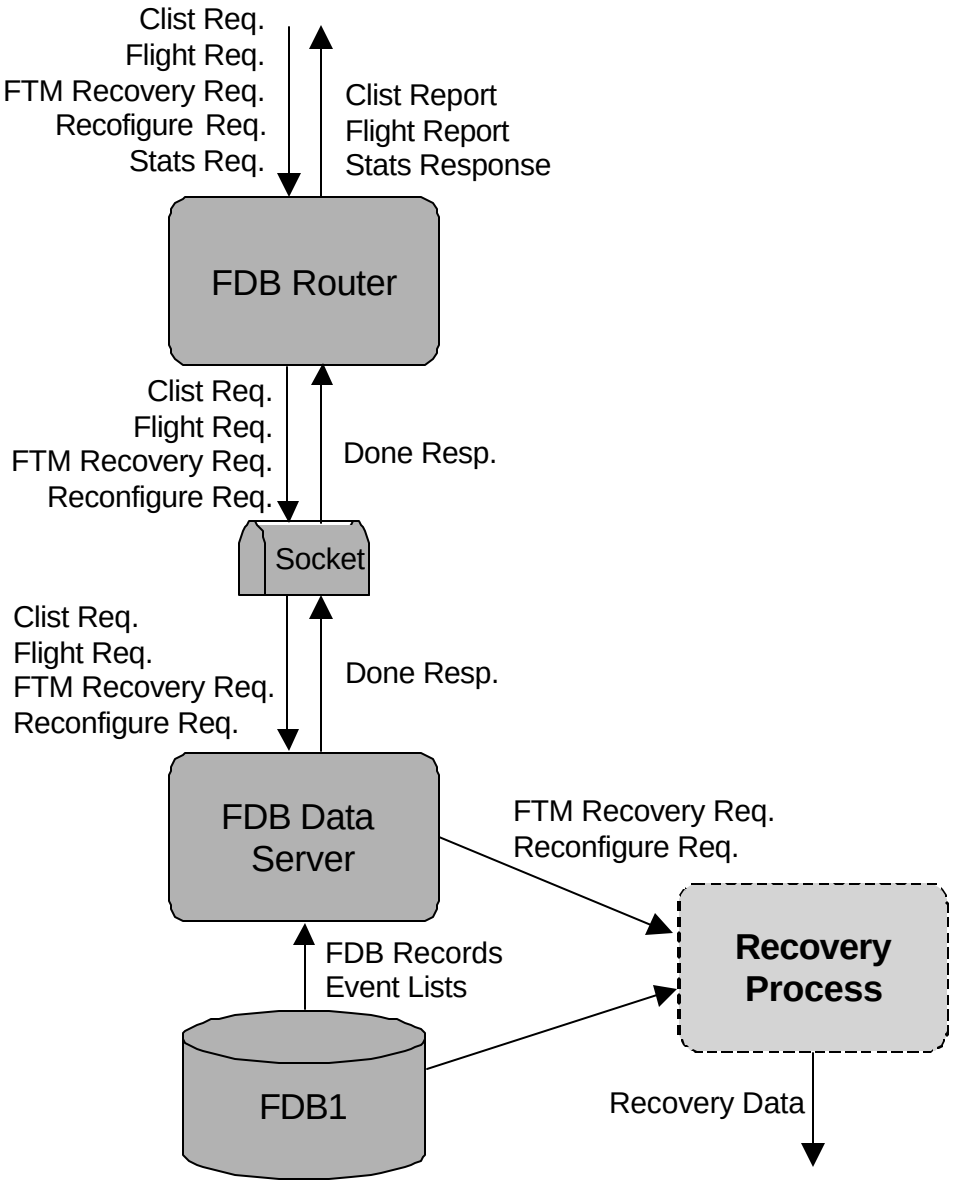

Figure 25-3. FDB Router/Data Server

Processing

When the *FDBR* begins execution, it creates the FDB router socket. This socket is used for all correspondence between the *FDBR* and the *FDB Data Servers*. Once the socket has been created, the *FDBR* synchronizes with the *FDB Data Server* by waiting until it has successfully connected to the socket. When the connection is complete, the *FDBR* connects to the node switch. The *FDBR* proceeds to its main processing loop where it waits on events either from network addressing or from the socket.

The *FDBR* processes five distinct event types from network addressing: clist requests, Fdata requests, ftm recovery requests, reconfigure requests, and stats requests.

(1) Center List (CList) Requests - Copy the source filename, restriction filename, clist output filename, and the airport flag into the message buffer. The message is sent via the *FDBR* socket to the *FDB Data Server*. The *FDB Data Server* responds with a Done message signaling the *FDBR* that the clist request has been fulfilled. The *FDBR* sends the clist to the destination address specified in the original request.

(2) Flight (FDATA) Requests - Copy the filename containing a list of flight IDs into the message buffer. Generate an empty response file. Send filenames to *FDB Data Server* and wait for response. The *FDB Data Server* accesses the FDB database and populates the response file. Send response file via network addressing to requestor.

(3) FTM Recovery Requests

(a) Fill-in - Decode a message from network addressing and forward the request to the FDB Data Server.

(b) Recovery Status - Decode recovery responses from the full recovery process and relay the status to the FTM if necessary.

(c) Recovery Stop - Check the queue of active recoveries and if the specified recovery is on the active queue, force removal from the queue by setting the expiration flag.

(d) Full Recovery - Decode the recovery request, insert it into the active or pending queue, forward the request to the FDB Data Server, and return status to the FTM.

(4) Reconfigure Requests

(a) Enable/Disable Full FTM Recoveries.

(b) Enable/Disable Mini FTM Recoveries.

(c) FDB-FDB Recovery - Decode the recovery request, forward the request to the FDB Data Server, and return status to the invoker.

(5) Stats Requests

(a) Level 1 – Send active full recovery stats to requester through network addressing.

(b) Level 2 – Send pending full recovery stats to requester through network addressing.

(c) Level 3 – Send completed full recovery stats to requester through network addressing.

Error Conditions and Handling

Fatal errors cause the *FDBR* to terminate. There are two errors that are considered fatal:

(1) Unable to create router socket. The *FDBR* retries one time before exiting.

(2) Unable to connect to node switch.

All non-fatal errors cause an error message to appear in the main process window of the ETMS operator's node.

25.9 FDB Data Server

Purpose

The purpose of *FDB Data Server* is to give other processes access to the FDB database. The *FDB Data Server* receives requests from the *FDB Router* and maps the FDB database to access flight information. The *FDB Data Server* processes requests and produces reports. The *FDB Data Server* also invokes recovery processes.

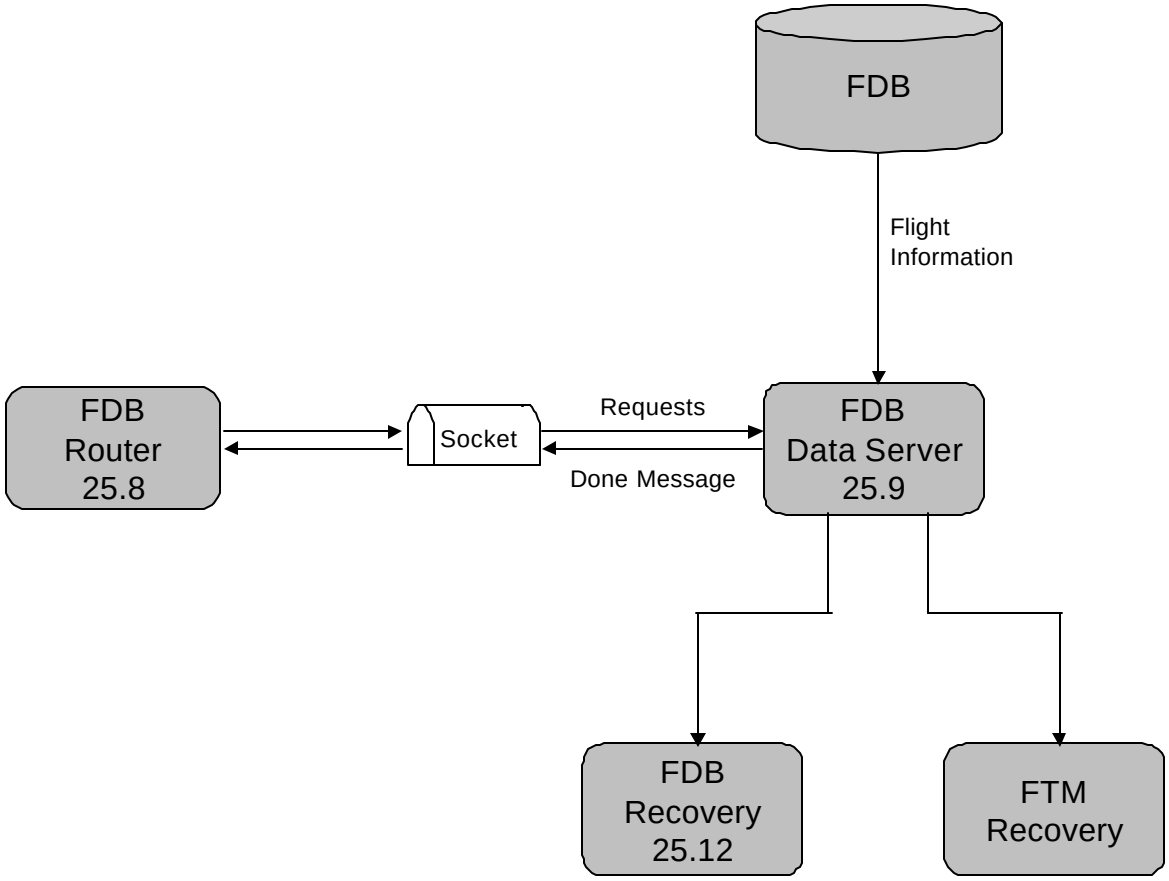

Figure 25-4. Data Flow of the FDB Data Server Process

Execution Control

The *FDB Data Server* is normally started by the *Nodescan* utility.

Input

The *FDB Data Server* requires fdb_data_server.params at startup. The fdb_data_server.params contains the *FDB Router* socket name, the location of the FDB database , and the path name of recovery process.

All other input is received through the *FDB Router* socket as follows:

(1) FDATA request – *FDB Data Server* receives the name of a file containing flight IDs.

(2) SDATA request – *FDB Data Server* receives the name of a file containing flight IDs.

(3) CLIST request – *FDB Data Server* receives clist file and restriction file names.

(4) FTM recovery request – There are full FTM recovery and fill-in FTM recovery requests.

(5) FDB recovery request – The full FDB cross-string recovery is for a slave FDB.

Output

The following is a list of the output files generated by the *FDB Data Server*:

(1) fdata_response.timestamp – contains departure and arrival airports, departure and arrival times, and departure and arrival centers. The file is located at ttm/fdb/programs/fdata_response.timestamp.

(2) pf_fdb_file.timestamp or file.*timestamp* – contains flight IDs, departure date, way points or proposed speed. These files are sent to ADR process.

(3) temp_clist.timestamp – contains airport information for flights. The file is located at ttm/fdb/dynamic_data/temp_clist.timestamp.

Processing

Processing begins with getting arguments from the parameters file and accessing the FDB database. When connecting to the *FDB Router* socket, the *FDB Data Server* receives FDATA, SDATA, CLIST, FTM recovery, and FDB recovery requests. For FDATA, SDATA, and CLIST requests, the *FDB Data Server* receives input files. After comparing with the FDB database, the results are written to output files. For FTM and FDB recovery requests, the *FDB Data Server* invokes child processes in background mode. FTM recovery has fill-in partial recovery and full recovery. FDB recovery has only full recovery. When complete, the *FDB Data Server* sends a done message to *FDB Router*.

Error Conditions and Handling

Errors cause an error message to appear in main process window of the *FDB Data Server*.

25.10 Route Relay

Purpose

The *Route Relay (Rt_relay)* process transmits ascii route transactions from the FDB to the NAS Dist process. Each time the *FDB Manager* receives a message which would cause the route of a flight to be updated, it enqueues this update buffer to a shared queue with the *Route Relay* process. The *Route Relay* process dequeues these update messages and sends them via the node switch to *nas.dist*.

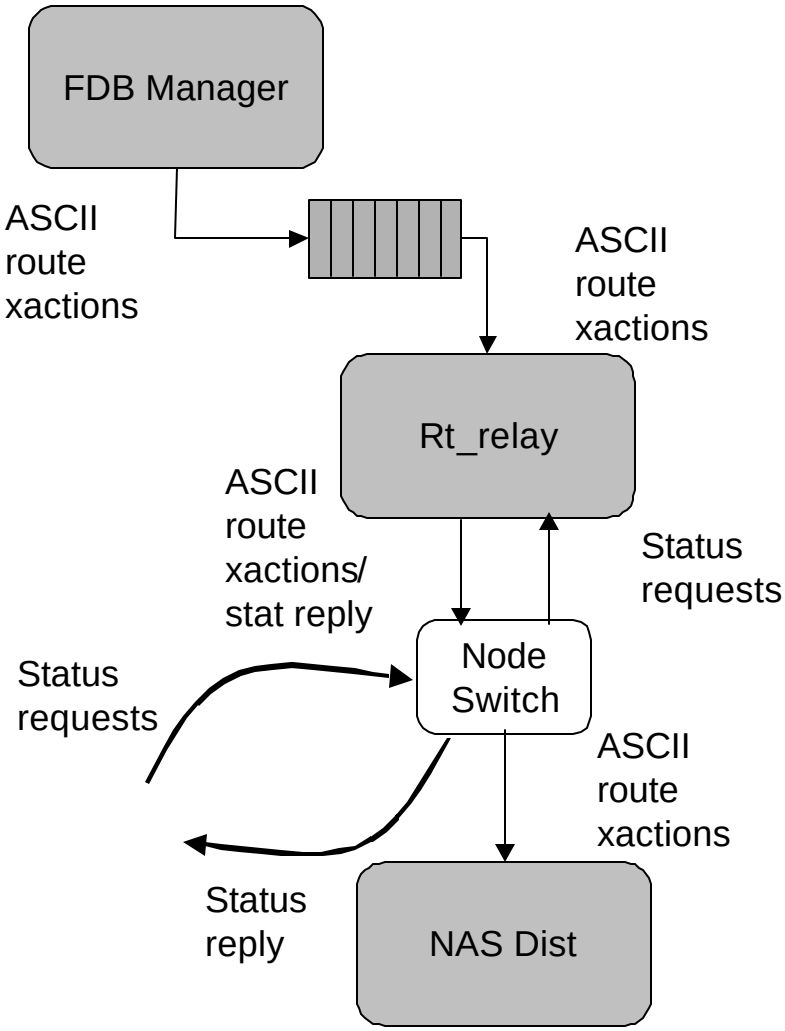

Figure 25-5. Data Flow of the Route Relay Process

Execution

The *Route Relay* process executes as a child of the *FDB Manager*. Once started, it executes continuously until its parent process terminates. It is strictly event driven and remains idle until either a status request is received or a route update message is queued by the *FDB Manager.* In the case of a route update message, no data transformation is required. The message is dequeued and forwarded to *NAS Dist*. For a status request, the transmission statistics are formatted into a readable message prior to transmission.

Input

The inputs to the *Route Relay* process are buffered ascii route updates and status requests.

Output

The *Route Relay* process sends ascii route updates to the *NAS Dist* process. It also responds to status requests with a buffer of transmission statistics.

Processing

The *FDB Manager* spawns the *Route Relay* process and creates the queue that is shared by both processes as part of its initialization sequence. The *Route Relay* process is spawned with two arguments: the shared queue number and a parameter file. The parameters file contains network related information which permits the *Route Relay* process to connect to the node switch. On initialization, the *Route Relay* process reads its associated parameter file and uses this information to connect to the node switch. Once connected, it maps an area of memory to be used as the shared queue with the *FDB Manager*. The *Route Relay* process waits for an event indicating that either a message has been enqueued by the *FDB Manager* or a status request has been sent.

When a status request is received, the *Route Relay* process responds with statistics detailing transmission characteristics (e.g., bytes dequeued or messages sent). When a route update is queued, the *Route Relay* process dequeues the message and forwards it to *NAS Dist*. Each time an event is processed, *Route Relay* validates its connection to the node switch, and if appropriate, reconnects.

Error Conditions and Processing

The *Route Relay* Process terminates in one of two ways:

- The *FDB Manager* (its parent process) terminates.
- The process is unable to connect to the node switch.

Recoverable errors cause a message to be printed to the process' output pad.

25.11 The FDB Distribution Process

Purpose

The *FDB Distribution* process transfers data transactions between the *FDPs* and connected client FTMs.

Execution Control

The *FDB Distribution* process is started by *Nodescan*. It reads a *parameter file* describing the process configuration and starts the *FDB Distribution Receiver* to get **TTM-FTM Transactions**. Once started, it runs continuously. If it halts, *Nodescan* restarts it. For more information on *Nodescan*, see Section 33.2.

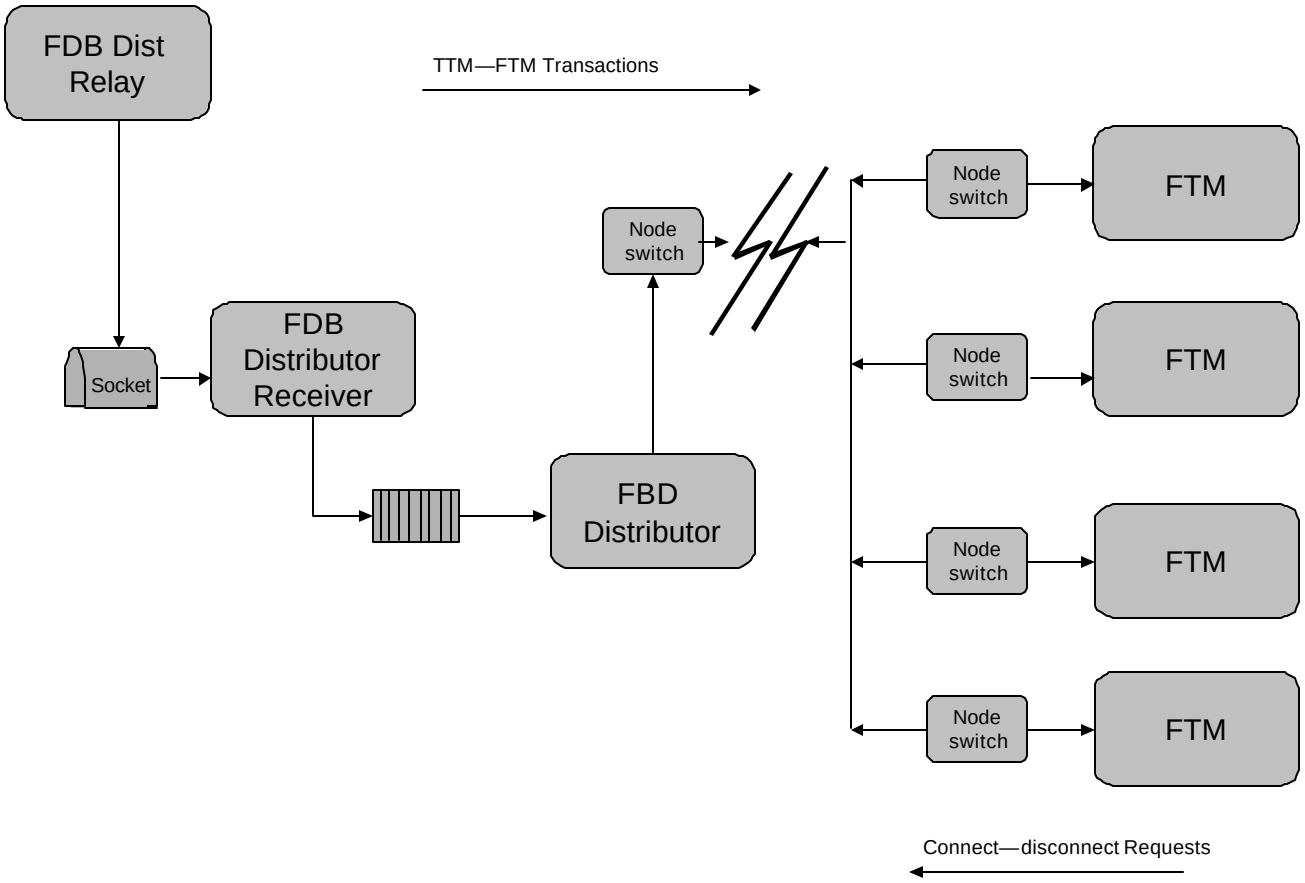

Figure 25-6. Data Flow of FDB Distribution

Input

The *FDB Distribution* process reads its parameters from a *parameters file*, provided in the program command line. The following is a list of parameters contained in the parameters file:

(1) FDB Distribution Receiver queue startup instruction that can be either **NEW** indicating a new receiver input queue is to be created or **OLD** indicating that the existing queue and its contents should be used. If the *queue* cannot be successfully opened, *FDB Distribution* ceases all processing, closes any *queue* that has been opened, and deletes the *memory_blocks* and *memory_stack_and_queues* files. After being restarted by *Nodescan*, *FDB Distribution* creates a fresh queue. *FDB Distribution* also opens and controls the process pads used to display processing statistics and error messages after which it establishes a node switch connection and registers as a provider.

(2) Number indicating which *FDB Distribution* process is to be started. The number 1 indicates the primary *FDB Distribution* process; any other number indicates the start of a secondary process which is a client to the primary process.

(3) ASCII string indicating the home site for the *FDB Distribution* process.

(4) ASCII string indicating the class name. If this is the secondary *FDB Distribution* process, an ASCII string indicating the primary process class name must follow the secondary class name on this line.

(5) Directory containing the executable programs to start up.

(6) Directory and file name of the site authorization file.

(7) ASCII string indicating the operator monitor class where all *FDB Distribution* process error messages will be sent.

(8) The receiver synchronization file name.

(9) FDB Distribution Receiver startup instructions. The line consists of the following three items:

- The file name of the FDB Distribution Receiver.
- Directory and file name of the FDB Distribution Receiver socket.
- The maximum number of queue blocks.

Additionally, the *FDB Distribution* process reads various optional process control switches from the command line. These may include the following:

- Option **-q <synchronization file name>** - This option directs the *FDB Distribution* process to receive input transactions from a local source.
- Option **-r <polling rate>** - This option directs the *FDB Distribution* process to "poll" all process connections at the specified polling rate, in seconds. The default value is 20 seconds.
- Option **-t <timeout value>** - This option directs the client *FDB Distribution* process to issue disconnect and reconnect requests to the primary *FDB Distribution* process at the specified timeout value, in seconds. The default value is 180 seconds.

After initialization, data input to the *FDB Distribution* process consists of **TTM-FTM transactions** queued up by the *FDB Distribution Receiver* process and generated by the *FDPs* and messages received through a network addressing switch (node switch). For a description of the contents of the input messages, see Data Structure Table 25-15 and following for **TTM-FTM transactions** and Section 33 for **node switch messages**.

Output

The *FDB Distribution* process buffers multiple **TTM-FTM transactions** by type **(**military or non-military) and sends them to the client FTMs. The buffering process does not alter the content of the individual transaction therefore individual input transactions are identical to individual output transactions.

Processing Overview

The *FDB Distribution* process transfers data transactions between the *FDPs* and connected client FTMs. The overall system can be configured with two *FDB Distribution* Processes; one as the

primary and one as a client to the primary.

At initialization, *FDB Distribution* reads the program parameter file name and optional switches from the command line. After it reads the parameters from the parameter file, it opens or creates an input queue for the *FDB Distribution Receiver* function and starts up the *FDB Receiver* as a child process. *FDB Distribution* determines whether to open an existing queue or create a new one based on the startup mode (*new* or *old*). If *FDB Distribution* is started with the *new* option, it creates a new input *queue*. Otherwise, it attempts to preserve the contents of the *queue* by opening the existing queue.

After initialization, *FDB Distribution* operates in a continuous, cyclic fashion. With each pass through the loop, it continuously monitors its child process (*FDB Distribution Receiver).* If the child process has stopped, it is restarted automatically. In addition to identifying incoming messages and processing them according to message type, *FDB Distribution* verifies its node switch connection, re-connecting as a provider if a disconnect has occurred. The three major processing modules within *FDB Distribution* are *FDB Distribution Receiver*, *Process FDP Message,* and *Process Node Switch Message*.

FDB Distribution Receiver The *FDB Distribution Receiver* process retrieves **FDP messages** from its socket and places them in the queue feeding *FDB Distribution.*

Process FDP Message The *Process FDP Message Module* dequeues **TTM-FTM transactions** from its input queue fed by the *FDB Distribution Receiver,* stores the transaction to either a military or non-military buffer and sends the buffers to connected clients. The buffers are sent when full or at a maximum wait of 30 seconds.

Process Node Switch Message The *Process Node Switch Message Module* handles messages received from the network through a node switch port. The messages processed are **returned messages** (messages sent by *FDB Distribution* that were undeliverable), **network informational messages** (messages requesting statistics or *FDB Distribution* specific identification information), **inter-fdbd messages** (messages between a primary *FDB Distribution* process and a client *FDB Distribution* process) and **connect/disconnect messages** from client FTMs.

Error Conditions and Handling

The following is a list of error conditions that cause program termination, with an error status:

(1) No parameters file name provided in command line, or unable to open parameters file.

(2) Unable to open or create the input queue.

(3) Specifying a command line switch without an associated value or specifying an unknown switch.

(4) Can't start the receiver.

(5) No distribution class named in the parameters file or the name given is too long.

(6) No site named in the parameters file or the name is too long.

(7) Unable to load the site authorization file.

(8) No operator monitor class named in the parameters file or the name is too long.

(9) No primary *FDB Distribution* process named when starting a client process.

(10) Unable to get timer event counter.

25.11.1 The FDB Distribution Receiver Module

The *FDB Distribution Receiver* is an instance of the generic receiver provided as a part of the Generic Buffering Package. For general information on this and other receiver processes, see Section 23.2

Purpose

The purpose of the *FDB Distribution Receiver* is to enqueue transactions generated by the FDP. The *FDB Distribution Receiver* buffers the messages in a manner that isolates *FDB Distribution* from any I/O processing, thus allowing *FDB Distribution* to process input transactions both asynchronously and chronologically.

Execution Control

The *FDB Distribution* process starts the *FDB Distribution Receiver* as a child process. This data-driven process runs continuously; if it fails, *FDB Distribution*, the parent process, restarts it using the pre-existent queue so that a minimum amount of data is lost. Non-fatal errors cause an error message to appear in the receiver window of the ETMS operator's node.

Input

FDB Distribution Receiver input consists of **TTM-FTM transactions** (Tables 25-15 through 25-21) generated by the FDP.

Output

The *FDB Distribution Receiver* output is identical to its input.

Processing

The *FDB Distribution Receiver* creates the socket to which the *FDB Dist Relay* connects. It also allocates memory for use as its output queue. As messages come into the socket, the *FDB Distribution Receiver* retrieves them and places them in its queue for use by the *FDB Distribution* process.

Error Conditions and Handling

For a complete description of possible errors generated by the *FDB Distribution Receiver,* see Section 23.2.

25.11.2 The Process FDP Message Module

Purpose

The purpose of *Process FDP Message* is to dequeue **FTM-TTM transactions** generated by the FDP, buffer the transactions and send them to the client FTMs. The messages are stored to a military or non-military buffer type.

Input

The input to *Process FDP Message* consists of five types of **TTM-FTM transactions.** They are **tz, time, route, cancellation**, and **position**. Each transaction type correlates to one or more **NAS** message type as shown below.

TTM-FTM transaction type	NAS message type
TZ	**TZ**
TIME	**AZ, DZ, EDCT**
ROUTE	**AF, FS, FZ, UZ, FA**
CANCELLATION	**RS, RZ, SI_CANCEL_FLIGHT, CONTROL_CANCEL**
POSITION	**TA, TO**

For a description of **TTM-FTM transactions** and **NAS message types,** see Tables 25-15 through 25-27 at the end of this Section.

Output

The output from *Process FDP Message* consists of buffers containing multiple **TTM-FTM transactions.**

Processing

The *Process FDP Message* module begins by checking the wait period (30 seconds) to send the buffers. If the time has elapsed and data has been stored, it sends the buffers via network addressing to its client FTMs and resets the timer. *Process FDP Message* then enters a loop dequeuing **FTM-TTM transactions** until the queue is empty. With each cycle through the loop, it increments the appropriate statistical counters, checks for and responds to **node switch** messages, loads the output buffers and when the buffers are full, sends them to the client FTMs.

Error Conditions and Handling

Process FDP Message handles the following six error conditions:

(1) **Can't dequeue a message** — *Process FDP Message* keeps trying to dequeue transactions until successful. An error message is written to the screen with each pass through the loop until a message is dequeued.

(2) **Bad message type** — An error message is written to the screen and processing continues.

(3) **Bad message size** — An error message is written to the screen and processing continues.

(4) **Unknown error when sending output buffer** — An error message is written to the screen and *Process FDP Message* goes into a loop resending the message until it is successful.

(5) **Bad output buffer size** — An error message and the dequeued message are written to the screen.

(6) **Output port full** — An error message is written to the screen and *Process FDP Message* goes into a wait loop until the port clears. When the port clears, held buffers are resent.

25.11.3 The Process Node Switch Message Module

Purpose

The purpose of *Process Node Switch Message* is to respond to messages received from network addressing.

Input

The input to *Process Node Switch Message* consists, in general, of two categories of node switch messages**.**

- **Returned messages** — Messages sent by the *FDB Distribution* which were undeliverable and were returned by the network addressing package.
- **Network informational requests** — Messages from other processes requesting information from *FDB Distribution*. These messages include **inter-fdbd messages** (messages between a primary and client *FDB Distribution* process) and **connect-disconnect requests** (messages that add or delete FTMs from *FDB Distribution*'s client list).

Output

The output from *Process Node Switch Message* is dependent upon the category of the input message and is explained in the *Processing* section that follows.

Processing

Process Node Switch Message begins by entering a loop that it does not exit until all messages have been retrieved. With each pass through the loop, *Process Node Switch Message* retrieves a message, determines the category of the message and acts based either on the network addressing return reason or the network addressing message type.

For *returned messages* with the following reasons, *Process Node Switch Message* attempts first to convert the site code to ascii and then to convert the class to ascii. If either conversion fails, it issues an error message and continues. If the primary *FDB Distribution* is receiving the *returned message*, it removes the source of the message from its client list. If it is the client *FDB Distribution*, it reconnects to the primary and resets the timing connection to the primary. If *Process Node Switch Message* receives a message that has a return reason other than the following, it waits one second, checks the port trigger value and cycles to the top of the loop.

- ERR_NET_BAD_NODE_ID
- ERR_NET_GATE_UNAVAIL
- ERR_NET_PCK_TIMEOUT
- ERR_NET_BAD_DEST_ADDRESS
- ERR_NET_BAD_RESOURCE_ID
- ERR_NET_QUEUE_CLEARED

Process Node Switch Message responds to *network informational messages* based on the type of request. These messages include requests from another *FDB Distribution* Process (connect-reconnect, client list and pulse request/reply), connect-disconnects from client FTMs and general user requests for statistical data.

For more information on network addressing, see Section 33.2.

Error Conditions and Handling

Process Node Switch Message writes an error message to the screen indicating the network process that failed and showing the source of the message, the destination of the message, or both as the error dictates. It continues processing, returning only when the port is empty. In the case of a full port, *Process Node Switch Message* issues an error, waits until the port is available and then continues standard processing.

25.12 The FDB Recovery Process

Purpose

The purpose of FDB cross-string recovery is to copy the entire flight database from one string to another. The FDB on the first string must be in Master mode and the FDB on the second string must be in Slave mode for cross-string recovery to occur. See the Design Issues in Section 25 for a description of Master and Slave modes.

Execution Control

When the *FDB Router* (Section 25.8) on the Master string receives an FDB Recovery request, it forwards this request to the *FDB Data Server* (Section 25.9) processes, which then spawns a new *FDB Recovery* process.

Figure 25-7 depicts the data flow for the *FDB Recovery* process.

Input

On startup, the *FDB Recovery* process receives the recovery parameters from the *FDB Data Server*. These recovery parameters identify the type of recovery to be performed, the time window within which all FDB records should be restored, the identity of the string being recovered, etc.

During recovery processing, the *FDB Recovery* process reads flight records and event lists from the flight database.

Output

The *FDB Recovery* process generates cross-string recovery messages containing flight database records, which are sent to the Slave FDB.

The *FDB Recovery* process tracks the total number of flight database records sent, the total number of errors that occurred while attempting to send records, and the total time required to complete cross-string recovery. These statistics are reported to the ETMS operator and to the *FDB Data Server*.

Processing

The *FDB Recovery* process determines the identity of the string whose flight database is to be replaced, and creates an output queue to which all of the cross-string recovery messages will be written. The *FDB Recovery* process then spawns a relay process to transfer messages from the queue to the socket of the *FDB Receiver* on the Slave string. (See Section 23.2 for a complete description of relay processes and message queues.)

The *FDB Recovery* process then reads through its entire flight database, one record at a time. If the record has not been marked as having been deleted from the database, then it is incorporated into a cross-string recovery message and written to the output queue.

The *FDB Recovery* process and its child relay process terminate when the cross-string recovery has completed.

See Figure 25-8 for a detailed sequential logic diagram of the *FDB Recovery* program.

See the Design Issues in at the beginning of this Section for a description of how the Slave FDB handles the cross-string recovery messages sent by the *FDB Recovery* process.

Error Conditions and Handling

Certain errors are fatal to the *FDB Recovery* process. This process terminates unsuccessfully if it cannot properly map the flight database files, or if the designated recovery string is not valid, or if it is unable to create the output queue and spawn a relay process. In these cases, error messages appear on the ETMS operator's node and the *FDB Recovery* process terminates.

Non-fatal errors cause an error message to appear in the receiver window of the ETMS operator's node.

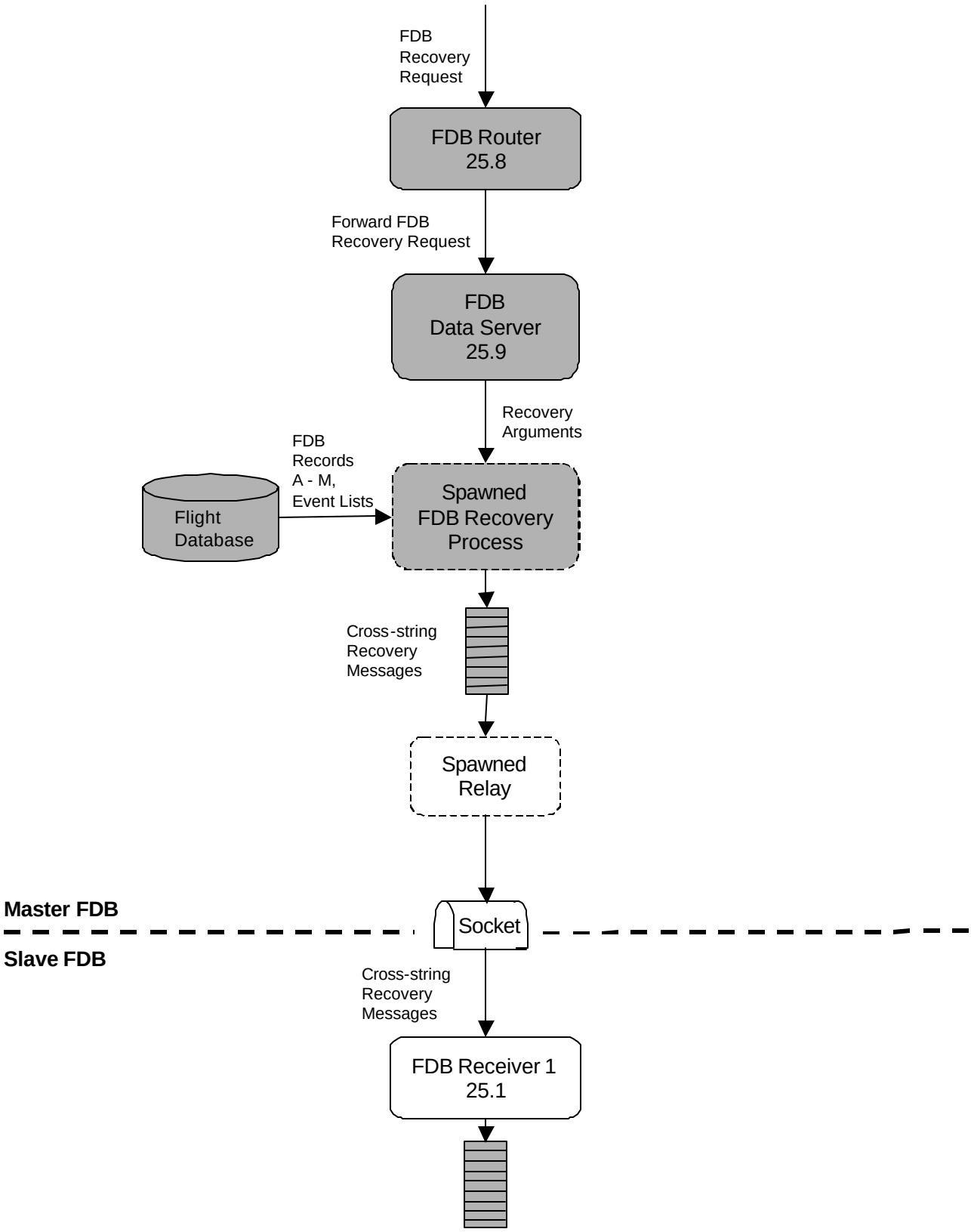

Figure 25-7. Data Flow of the FDB Recovery Process

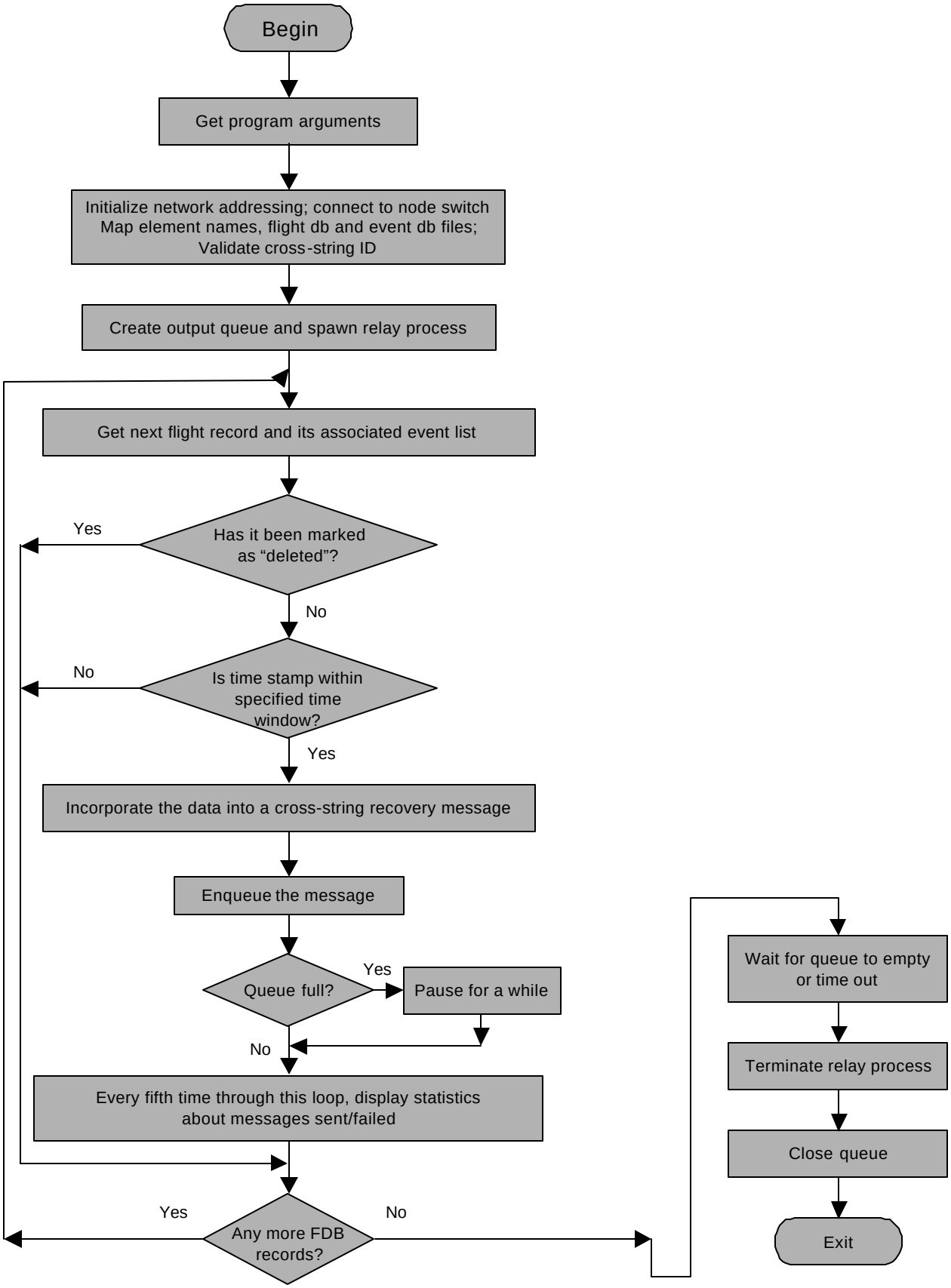

Figure 25-8. Sequential Logic for the FDB Recovery Process

25.13 The FTM Recovery Process

Purpose

The purpose of the *FTM Recovery* process is to restore part or all of the FTM database at a field site from the flight database at the central site. There are two main types of FTM recovery: full recovery, in which restores the entire database, and fill-in recovery, which restores information for specifically requested flights.

Execution Control

When the *FDB Router* (Section 25.8) receives an FTM Recovery request, it forwards this request to the *FDB Data Server* (Section 25.9) process, which then spawns a new *FTM Recovery* process. The two *FTM Recovery* processes behave in an identical manner.

Figure 25-9 depicts the data flow for the *FTM Recovery* process.

Input

On startup, the *FTM Recovery* process receives the recovery parameters from the *FDB Data Server*. These recovery parameters include

(1) The recovery type (full or fill-in)

(2) The maximum number of FDB records allowed to be recovered

(3) The maximum number of data blocks per flight

(4) The pathnames of the FDB and EVDB files to be recovered

(5) The FTM and FTM Coprocessor network addresses

(6) The sequence number

(7) The recovery time

(8) The pathnames of the Element Names and ARTCC Pairs files

(9) The pathname of the sitefile

(10) The flight offset

(11) The number of flight records to be recovered (fill-in requests only)

(12) A list of flight IDs and their corresponding TDB indices (fill-in requests only)

(13) A start/stop time window (for full recoveries only; only flight database records whose time stamps are within this window will be recovered)

During recovery processing, the *FTM Recovery* process reads flight records and event lists from the flight database. The format of these records is described in Section 25.14.

Output

The *FTM Recovery* process generates recovery messages containing flight database records, which are sent to the FTM, and status messages, which are sent to the *FDB Router*.

The *FTM Recovery* process tracks the total number of flight database records sent, the total number of errors that occurred while attempting to send records, and the total time required to complete FTM recovery. These statistics are reported to the ETMS operator and to the *FDB Data Server*.

Processing Overview

The *FTM Recovery* process invokes its *Initialize* routine to create its output files, read the input parameters, determine the type of recovery to be performed, and map the flight database files. For full recoveries, the *FTM Recovery* process then invokes the *process_full_recovery* routine (see Section 25.13.1); for fill-in recoveries, it invokes the *process_data_recovery* routine (see Section 25.13.2).

See Figure 25-10 for a sequential logic diagram of the *FTM Recovery* process.

See Section 18 for a description of how the FTM handles the recovery data blocks sent by the *FTM Recovery* process.

Error Conditions and Handling

Certain errors are fatal to the *FTM Recovery* process. This process terminates unsuccessfully if it cannot properly map the flight database files, or if the designated FTM network address is not valid, or if it is unable to create the output queue and spawn a relay process. In these cases, error messages appear on the ETMS operator's node and the *FTM Recovery* process terminates.

Non-fatal errors cause an error message to appear in the receiver window of the ETMS operator's node.

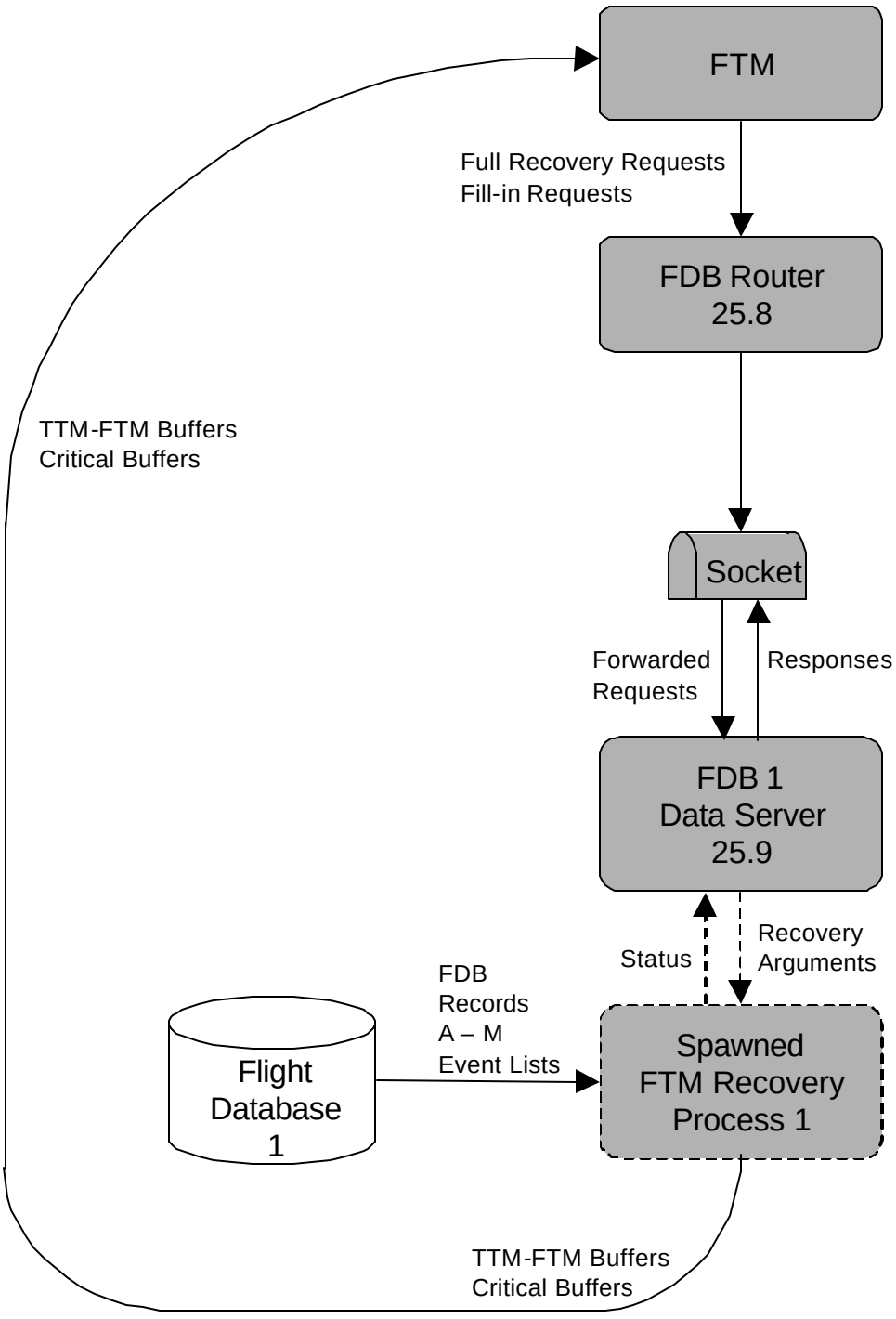

Figure 25-9. Data Flow of the FTM Recovery Process

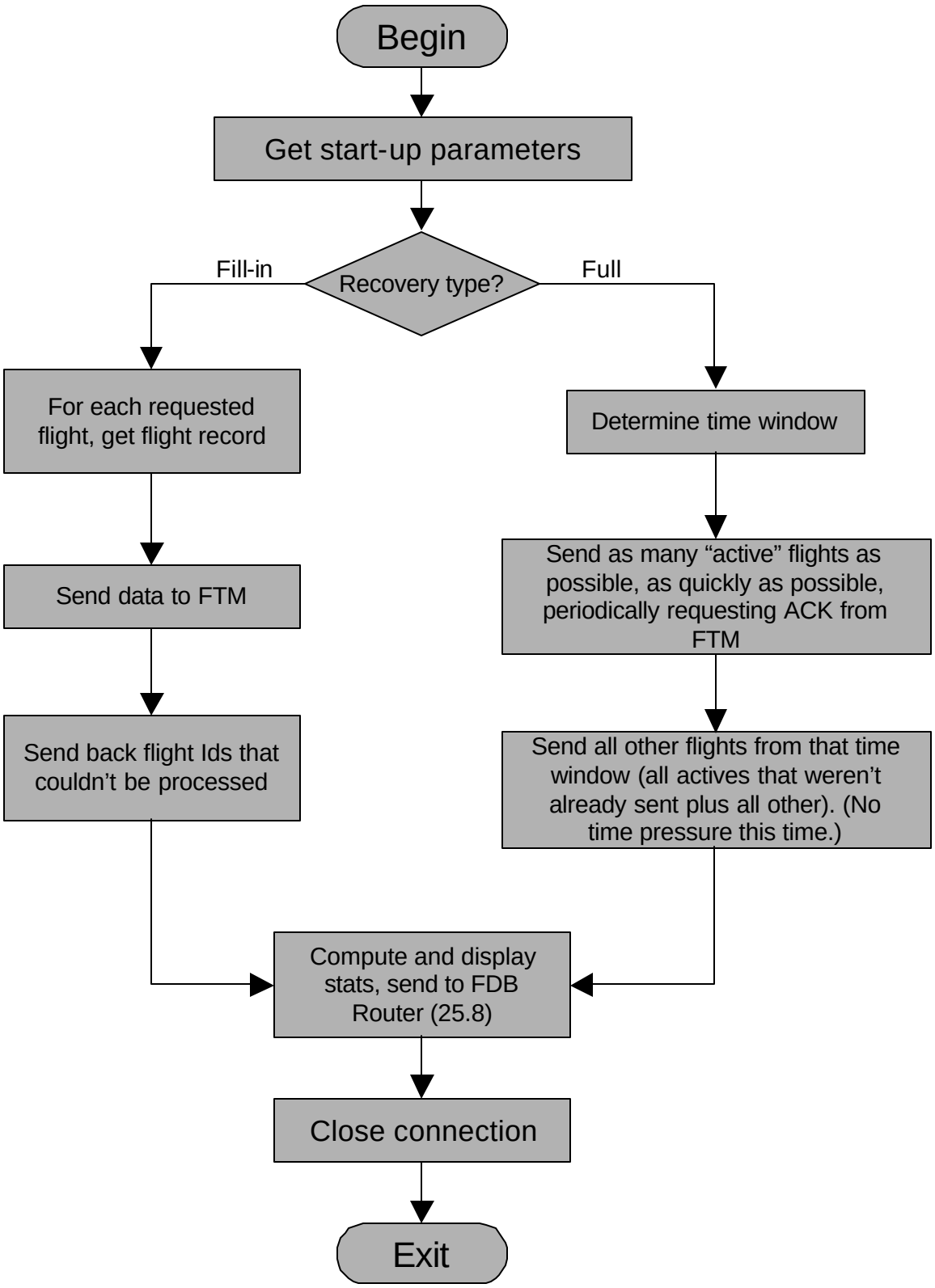

Figure 25-10. Sequential Logic for the FTM Recovery Process

25.13.1 The process_full_recovery Routine

Purpose

The purpose of the *process_full_recovery* routine is to retrieve all of the flight records and event lists in the flight database whose time stamps fall within a specified time window, and to send these records to the FTM that requested them. The time window can be as small as 15 minutes, or can cover the entire database.

Input

The *process_full_recovery* routine receives a full-recovery request record. This record identifies the network address of the requesting FTM and the requested recovery time window.

Output

The *process_full_recovery* routine sends several status records to the *FDB Router*. These status records indicate the progress of the recovery.

Processing

The *process_full_recovery* routine performs the following steps:

- Send a status message to the *FDB Router* indicating that FTM recovery has been initiated
- Call the *extract_valid_indexes* routine (Section 25.13.1.1) to retrieve and send the flight records for active flights
- Send a status message to the *FDB Router* indicating that the first part of the FTM recovery is complete
- Call the *send_remaining* routine (Section 25.13.1.2) to retrieve and send the flight records for all remaining flights
- Send a status message to the *FDB Router* indicating that FTM recovery has been completed
- Display a summary of the recovery process statistics on the screen

25.13.1.1 The *extract_valid_indexes* Routine

Purpose

The *extract_valid_indexes* routine is a first-pass attempt to send as many critical flight database records as possible to the FTM, as quickly as possible. Only records for active flights are sent in this first pass.

Input

The *extract_valid_indexes* routine receives a full-recovery request record, which identifies the network address of requesting FTM and the requested recovery time window. This routine also reads data from the flight database (described in Section 25.14).

Output

The *extract_valid_indexes* routine sends status records to the *FDB Router*, which indicates the progress of the recovery, and sends requests for acknowledgement records to the FTM. This routine also updates a global status record, which keeps track of the total number of records sent to the FTM, the total number of flights of each type (active, controlled, scheduled, etc.), the total number of bytes sent to the FTM, and the number of acknowledgement requests sent and received.

Processing

The *extract_valid_indexes* routine examines each record in the flight database. If the record has not been marked as "deleted", and if its time stamp falls within the specified time window, then it will be sent to the FTM. If the flight's status is **active** and this process is not currently waiting for an acknowledgement message from the FTM, then *extract_valid_indexes* calls the *packsend_critical* routine (see Section 25.13.1.2.1) to pack the flight record and its associated event list into a buffer, and to send the buffer to the FTM if it is full. If the flight's status is not **active**, or if this process is waiting for an acknowledgement message from the FTM, then the flight record is not sent yet; its FDB address is saved to be sent later.

After a certain number of blocks have been sent to the FTM (defined by the constant **BLOCKS_BETWEEN_ACKS**), a status message is sent to the *FDB Router* process indicating that the FTM recovery is still in progress, and an acknowledgement-request message is sent to the FTM. No more recovery data blocks will be sent to the FTM until an acknowledgement message has been received from the FTM. (The *extract_valid_indexes* routine is not idle while waiting for the acknowledgement; it continues searching through the flight database for records to be included in the recovery, and it saves the addresses of these records to be sent later.)

The *check_fmtr_port* routine (see Section 25.13.1.2.2) is called regularly to look for and process incoming messages from the FTM.

25.13.1.2 The send_remaining Routine

Purpose

The *send_remaining* routine is the second pass through the flight database. All flight database records within the time window that were not sent by *extract_valid_indexes* in the first pass are sent here.

Input

This routine receives a copy of the full-recovery request record, which identifies the network address of requesting FTM and the requested recovery time window. This routine also receives a list of FDB addresses that were saved by *xtract_valid_indexes* in the first pass; records at those addresses will be sent to the FTM. This routine also reads data from the flight database (described in Section 25.14).

Output

The routine sends status records to the *FDB Router*, which indicates the progress of the recovery, and sends requests for acknowledgement records to the FTM. This routine also updates a global status record, which keeps track of the total number of records sent to the FTM, the total number of flights of each type (active, controlled, scheduled, etc.), the total number of bytes sent to the FTM, and the number of acknowledgements requests sent and received.

Processing

The *send_remaining* routine retrieves the flight records and event lists for all flights whose FDB addresses were saved by the *extract_valid_indexes* routine, and sends them to the FTM. (These include those records for **active** flights that could not be sent by *extract_valid_indexes* because this process was waiting for an acknowledgement from the FTM, and all flights of all other types.) Similar to *extract_valid_indexes*, *send_remaining* requests an acknowledgement from the FTM after a certain number of blocks has been sent; unlike *extract_valid_indexes,* however, it does no other processing while waiting for the acknowledgement.

The *send_remaining* routine sends flight records in order of flight status. They are sent in the following order:

(1) active

(2) controlled

(3) filed

(4) scheduled

(5) cancelled

(6) completed

(7) decontrolled

(8) ascending

(9) cruising

(10) descending

(11) no status

(12) error

The *check_fmtr_port* routine (see Section 25.13.1.2.2) is called regularly to look for and process incoming messages from the FTM.

25.13.1.2.1 The packsend_critical Routine

Purpose

The purpose of the *packsend_critical* routine is to pack flight database records into a buffer, and to send the buffer to the FTM when it is full.

Input

The *packsend_critical* routine receives the flight database record to be packed, the status of the flight (scheduled, files, active, etc.), and the network address of the FTM to which the buffer of packed flight database records is to be sent.

Output

The *packsend_critical* routine sends a buffer filled with packed flight database records to the FTM. This routine also updates a global status record, which keeps track of the total number of bytes sent to the FTM. It also indicates to its calling routine whether or not it was able to successfully pack the record, and whether or not it sent the buffer to the FTM.

Processing

The *packsend_critical* routine packs a flight record and its associated event list into the format expected by the FTM. Data for flights whose status is **active, controlled,** or **filed** are packed by the *packftmblk* routine; all others are packed by the *packftmrcvr* routine (see Section 25.13.1.2.1.1).

The *packsend_critical* routine maintains a buffer of the packed FTM records. If there is room in the buffer for the newly packed record, then it is added to the buffer. If there is not enough room, then *packsend_critical* sends the buffer to the FTM, clears it, and stores the newly packed record into the empty buffer.

25.13.1.2.1.1 The packftmblk and packftmrcvr Routines

Purpose

The purpose of the *packftmblk* and *packftmrcvr* routines is to reformat flight database records into the format expected by the FTM.

Input

These routines receive as input the flight database record to be packed, the status of the flight, and the network address of the FTM to which the data is to be sent.

Output

The *packftmblk* routine packs the flight database record into the format shown in the TTM-FTM Block Transaction table (see Table 25-15); *packftmrcvr* packs the data into the format shown in the TTM-FTM Recovery Transaction table (see Table 25-17).

Processing

Both of these routines first pack the FDB-FTM data block with a departure date, time stamp, flight status, altitude, and speed. (For altitude, the routines use reported values if they exist; otherwise they use proposed values. For speed, they use proposed values if they exist; otherwise they use reported values.)

The *packftmblk* routine (which was called for flights with a status of **active, controlled,** or **filed**) then determines the next lat/long point from the event list and packs that into the record; it also adds the six departure/arrival times (scheduled/proposed/actual/estimated/controlled/original), arrival fix data, waypoints, sectors, fixes, airways, and centers. The *packftmrcvr* routine packs only two of the six departure/arrival times (controlled and original).

Both routines pack the estimated times of departure and arrival and the estimated time en route, the departure and arrival airports and centers, the aircraft type, category, and weight class, and the arrival fix data.

25.13.1.2.2 The check_ftmr_port Routine

Purpose

The purpose of the *check_ftmr_port* routine is to look for and process messages from the FTM.

Input

This routine is given the network address of the FTM by its calling routine; it receives various types of messages from the FTM.

Output

This routine updates the acknowledgment-counter when an acknowledgment message is received from the FTM, and tells its calling routine that it is no longer waiting for an acknowledgment. This routine also updates the global FTM recovery status record, and creates status records for the *FDB Router*.

Processing

The *check_ftmr_port* routine checks for messages from the FTM. (It is a non-blocking check; if there are none, it does not wait for one.) For each FTM message in the queue, the *check_ftmr_port* routine checks the message status. If it is an error message, a message is displayed for the operator; depending on how serious the error is, a fatal-error status block may be formatted and sent to the FDB Router, and this process may terminate. If it's not an error message, then the message type is checked, and processing occurs as follows according to message type, as follows:

- **net$_t_msg_data** — A message is displayed notifying the operator that this message was received.
- **net$_t_site_reconnect** — This routine calls *net$_inq_my_address* to check this process's network address.

- **ftm$_t_ack_ready_receive** — The acks-received counter is incremented, and the calling routine is told that an FTM acknowledgement has been received.
- **ftm$_t_recovery_stop** — If the source address class is FDBR, then a fatal status message is sent to the FDB Router and this process terminates. If the source address class is not FDBR, then this type of message is ignored.
- **ftm$_t_recovery_resend** — Messages of this type are ignored.
- **ftm$_t_recovery_status** This is a status request from the FDB Router. The *check_ftmr_port* routine creates a record reflecting the current status and calls *nwa_send_message_pri* to send to the FDB Router.
- **nwa_fdbd_to_fdbd** — This type of message is treated as an acknowledgement from the FTM. (See **ftm$_t_ack_ready_receive** above.)
- **ftm$_t_give_status_lev** — This routine calls *handle_stats1* to format and send a status message to the requesting process. The status message includes the elapsed time, the network addresses of the FDB and FTM, the total number of flights, data blocks, and bytes to be sent to the FTM, the number sent so far and the number remaining to be sent, and the number of acknowledgments requested and received from the FTM.

25.13.2 The process_data_recovery Routine

Purpose

The purpose of the *process_data_recovery* routine is to start the fill-in FTM recovery, retrieve the flight records and event lists for specifically requested flights, and to send these records to the FTM that requested them.

Input

List of flight IDs & their corresponding TDB indices; network address of requesting FTM; flight database

Output

The *process_data_recovery* routine sends the network address of the FTM to be recovered, and the list of requested flights, to the *respondtoftm* routine.

Processing

The *process_data_recovery* routine calls the *respondtoftm* routine, telling it that the message type is **ftm$_t_ttm_ftm**.

25.13.2.1 The respondtoftm Routine

Purpose

The purpose of the *respondtoftm* routine is to send specifically-requested flight database records to the FTM.

Input

List of flight IDs and their corresponding TDB indices; network address of requesting FTM; flight database

Output

FDB data blocks for flights that were found; a list of returned flights for those flights with incompatible flight IDs

Processing

For each requested flight, the *respondtoftm* routine converts the flight's TDB index to the corresponding FDB address and gets the FDB record from that address. It compares the flight ID of the retrieved record with that specified in the input parameters from the FTM. If the flight IDs match, then *respondtoftm* calls the *packftmblk* routine (see Section 25.13.1.2.1.1) to pack up the flight record and its event list into an FTM data block. If the flight IDs do not match, then it is added to the return-to-FTM record and ignored. When the FTM data block is full, it is sent to the FTM and a new FTM block is started.

After all of the requested flights have been processed, if there is any data remaining in a partially-filled block, it is sent to the FTM. The list of flights whose flight IDs didn't match their TDB indices, if any, is also sent back to the FTM.

25.14 The FDB Manager Process

Purpose

This is the main **flight database** process. It is responsible for keeping an up-to-date entry on every flight from those that landed 12 hours ago to those that will be taking off 12 hours in the future. It is also effectively a switching station for flight information, receiving all messages, then forwarding pertinent updates to a variety of final destination processes.

If it is running in master mode, it receives parsed NAS messages (flight plans, position updates, etc.) from the *Parser*, matches each message to the database entry it concerns, updates the entry with the data from the message, then distributes updates relating to the new information around the ETMS system. If it is running in slave mode, it accepts database update messages from the master *fdb_manager*, then plugs them into its database, and generates updates for processes connected to it for data.

In addition to this, each *fdb_manager*

- Maintains a picture of the current high altitude winds to use when predicting flight performance.
- Invokes and monitors all of its child processes (receiver and relays).
- Monitors the status of a communication channel to the network addressing package.
- Coordinates the current master/slave setup between strings.

Design Issue: Two Files (fdb and evdb) Make Up the Flight Database

The FDB consists of two mapped files: the **fdb** and the **evdb**. Each of these files is made up of records that contain different, yet related, information about each flight.

Each **fdb** record holds state information for one flight. Flight state information includes general flight information (aircraft type, flight ID, etc.), departure and arrival times, and current flight status (position, speed, etc.). Refer to 25-10 for a detailed description of the **flight_db_type** record.

In addition to flight state information, each flight may have route information constructed from a message containing a field 10 (flight plan). This route information is kept in the form of an event list - a list of the NAS events (fix crossing, sector entry, sector exit, jet route tracking) that the flight is expected to generate (and the times each event is expected). This event list is kept in a separate mapped file called the **evdb**. The two mapped files are linked by means of a field in each **fdb** record containing the address of a record in the **evdb**. The relationship of the two files is depicted by Figure 25-11.

Each **evdb** record is a block of a set number (here, the constant **eventsperblock**) of events, where each event is a record of the type **erect** (see 25-11). A variable-length event list is stored as one or more of these blocks. For example, if **eventsperblock** is 30, a list of 50 events would be stored in two **evdb** records. The first record would contain the first 30 events, and the second record would contain the remaining 20 events, followed by 10 empty events. The two **evdb** records are connected by the storage of the second record's address in a field in the first record. The connection of event data blocks is also illustrated in Figure 25-11.

The *fdb_manager* process allocates records in the **fdb** and **evdb** by treating each map file as a circular buffer of records. It maintains two variables (in the **crashfile**), which indicate the address of the last record allocated. As **fdb** and **evdb** records are allocated, the *fdb_manager* process increments these addresses until it reaches the end of the file. At this point, it resets the address to the beginning of the file and starts over again. In order to prevent reusing records which are being used by another flight, both **fdb** records and **evdb** blocks contain a field which indicate they are in use. When flights are deleted from the database, their event blocks and flight records are marked to indicate that they are available for use by another flight.

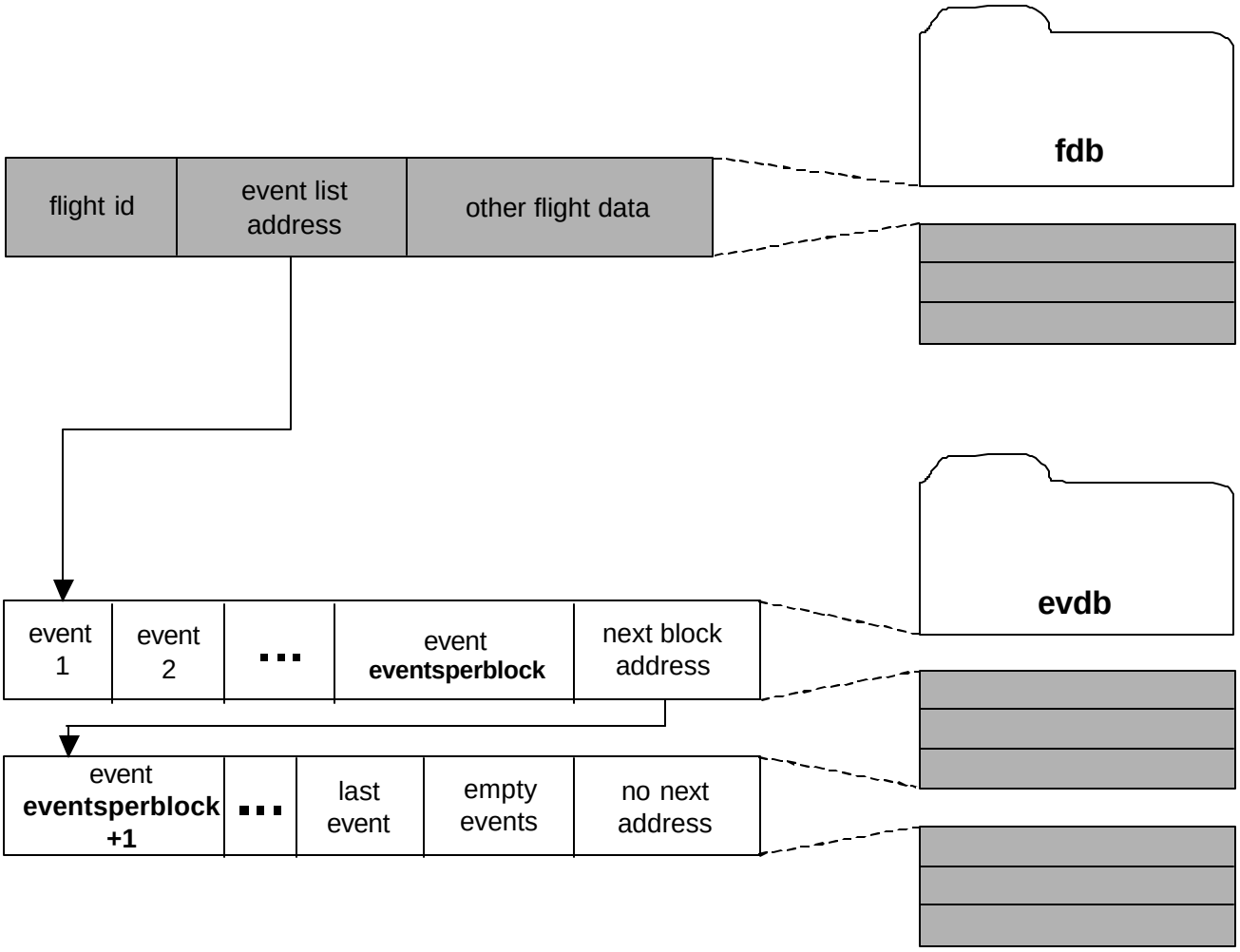

Figure 25-11. Flight Database Breakdown into Mapped

Execution Control

The *fdb_manager* process is invoked and monitored by an external process called *Nodescan*. *Nodescan* starts this process when the node on which the *fdb_manager* process is to run is booted and if the process halts, *Nodescan* will restart it. For more information on the *Nodescan* process, consult Section 33.

Input

The *fdb_manager* process receives two major types of input: start-up and initialization data in the form of parameters files and switches, and the static and dynamic data the *fdb_manager* process uses in its continuous processing. The process start-up and initialization data includes the following parameters files and optional run-time switches:

(1) The main start-up parameters file from which the *fdb_manager* process obtains database configuration and run time parameters. This text file contains process configuration information and the names of additional parameter and data files for use by the *fdb_manager* process and its child processes. Some of the items

contained in this parameter file include

(a) The path name and screen placement for process output pads.

(b) The locations and sizes of the flight database mapped files.

(c) Child process (receiver and relay) program and parameter file names.

(d) Communication login parameters.

(e) Grid-winds data file parameters.

(f) File names for ground time data.

(g) A list of valid first characters for aircraft flight IDs. This list is used to distribute flights over multiple *fdb_manager* processes, when necessary for increased processing speed. Each process maintains only those flights whose flight IDs begin with characters listed for that process.

(h) The name of the parameter file containing the location of the aircraft dynamics database.

(i) The name of the TZ processing parameters file, which provides parameters such as the time interval between messages and the maximum allowed deviation from the route for processing of position update (TZ) messages.

(2) The **-d** switch, which instructs the *fdb_manager* process to send information on flights removed from the database or the TDB to text files. Two parameters in the main parameter file define path names for these output files. This switch is not normally used.

(3) The **-t** switch, which tells the *fdb_manager* process not to send time-of-day transactions to the TDB. This switch is activated on all processors except one, when there is more than one *FDP* in a given ETMS string.

(4) The **-s** switch, primarily of use during process debugging, tells the *fdb_manager* process to use time stamps from all message types as the current time value sent to the *Time Out Flights From FDB* and *Remove Late Departures From TDB* modules. Normally, only time stamps from NAS messages are used for this purpose.

The *fdb_manager* process also uses the following types of input:

(1) Static data contained in files:

(a) Aircraft dynamics data –– This data describes the dynamics of an aircraft based on a variety of factors; a most important one is airplane type.

(b) Element name –– This file is used in generating route information. It maps a NAS element's internal index and type code into its name.

(c) Ground Times –– This data consists of historic ground times based on particular flight legs as well as by flight categories. The ground time is the

difference between a flight's proposed push-back time and its wheels-up time.

(d) ARTCC name/character pairs –– Each pair consists of the name of each ARTCC and its corresponding single character designator. Used when compiling ARTCC information during route transaction generation.

(2) Dynamic non-flight data –– This is data which may change during execution, but which does not involve information about a particular flight.

(a) High altitude winds data. This data is used in predicting flight times.

(b) Messages from the network addressing package. These messages could be

- Notification that a new winds file has arrived to be read in.
- Master/slave coordination message.
- Reconfigure command.
- Statistics request.

(c) NAS time messages –– Used as the clock for the *fdb_manager* process, these messages are actually the time stamps assigned to the NAS messages as they are received by the ETMS.

(d) EDCT FA advisories –– These messages indicate that an FA delay is assigned to a certain airport for a certain time limit.

(e) Master/Slave coordination messages –– Used to control which string is running in master mode.

(3) Dynamic, non-NAS flight data –– This is data that involves particular flights, but which is not one of the standard NAS messages:

(a) Parsed EDCT messages –– These messages, generated by the *EDCT Server*, supply information about ground delays issued by Air Traffic Control System Command for individual flights.

(b) Parsed flight schedule messages –– These messages, sent to the *fdb_manager* process by the *Schedule Database Processor* via the *Parser*, provide the *fdb_manager* process with information regarding scheduled flights. The two types of schedule messages are: Scheduled Flight Plan (FS) and Schedule Cancellation (RS).

(c) Feedback messages (FA) –– These messages are returned by the *Parser* in response to an FA from the *FDB Manager*. The FA from the *FDB Manager* is a combination of an AF message from the *Parser* with information about a matching flight in the flight database.

(4) Parsed NAS messages –– These contain the flight state and route data that have been drawn from NAS messages by the *Parser*. The data available in these messages can be found in Section 6. The standard NAS message types are

(a) FZ –– Flight plan.

(b) DZ –– Departure message (flight is in the air).

(c) TZ –– Position update on active flight based on a radar return.

(d) TO –– Position update on active flight flying over an ocean (call-in).

(e) AF –– Amended flight plan.

(f) UZ –– ARTCC boundary crossing (includes route data).

(g) AZ –– Arrival message (flight has landed).

(h) RZ –– Flight cancellation message.

Output

The *fdb_manager* process produces the following types of output for use in other parts of the ETMS:

(1) TTM-FTM transactions –– For each flight message received, the *fdb_manager* generates a transaction for the flight that is sent to the *fdb_dist* process (to be forwarded to any *Flight Table Manager (FTM)* processes that are connected.) These transactions are binary, rather than in ASCII format.

(2) TDB transactions –– Provides the TDB with the current time, flight status, flight route events, flight times, and instructions to time out late-departing flights.

(3) Feedback (FA) messages –– These messages are sent to the *Parser* in response to most flight plan amendment (AF) messages received by the *fdb_manager* process. These messages provide the *Parser* with information needed to parse the AF that may not be contained in the original AF message.

(4) Deleted flight information messages –– Whenever a flight is deleted from the Flight Database, that flight's information is sent to the *Ground Time Prediction Server* via the *DAS Relay*.

(5) Updates on controlled flights –– The *EDCT* process informs the *fdb_manager* when a control exists on a flight. The *fdb_manager* has to inform the *EDCT* process when a flight plan (FZ) is received on a controlled flight. The *fdb_manager* also sends messages to the *EDCT* process when a controlled flight goes active, or if one is cancelled.

(6) V4 routes –– In Version 4.0 of ETMS, the *fdb_managers* forwarded information on flight plans (routes) to various places. One of these places was the data feed for the ATA (Air Transport Association). As part of the FAA/ATA agreement, we must continue to provide these V4.0 format messages. These messages are referred to as RT messages.

(7) Cross-string updates –– If an *fdb_manager* is running in master mode, it may be sending database update messages to a *fdb_manager* on another string.

(8) Master/Slave Coordination messages –– Used to coordinate which string is in charge.

(9) Responses to statistics requests.

In addition, the *fdb_manager* process sends message statistics and non-fatal error messages as output to its main transcript pad. This output is used in the monitoring of the *fdb_manager* process. The statistics display includes, among other things, how many messages of each type were received, the rate and type of matching to existing flights, how many TDB transactions were generated, and how many non-fatal errors were generated while processing position update (TZ) messages.

Since the *fdb_manager* process is designed to run continuously, and since the maximum disk space used by these output pads is limited by the operating system, a mechanism for monitoring the size of the output pad is required. The *fdb_pad_check* routine accomplishes this task by examining the size of the output pad once every fifteen minutes. When the pad size becomes larger than one megabyte, *fdb_pad_check* calls the routine *fdb_pad_create*, which closes the old output pad and creates a new one.

Processing Overview

Figure 25-13 illustrates the data flow between the four main modules which make up the *fdb_manager* process: *Process Flight Messages*, *gridwinds_read*, *Time Out Flights From FDB*, and *Remove Late Departures From TDB*. This section contains a brief description of the specific processing that takes place within each module. Figure 25-12 gives an overview of the *fdb_manager* processing. The sections that follow contain a detailed description of each module, together with specific data flows and sequential logic diagrams.

The module *Process Flight Messages* receives parsed NAS, flight schedule, and feedback (FA) messages from the *Parser* as well as EDCT messages from the *EDCT Server*. *Process Flight Messages* first attempts to match the incoming message with one of the flights already in the **flight database**. If the match is successful, *Process Flight Messages* uses the data contained in the message to update the existing flight data in both the **fdb** and **evdb**. If no match is found, *Process Flight Messages* assumes this is a new flight and allocates a new flight record and the appropriate number of event blocks, adding the data contained in the message to the databases. When necessary, *Process Flight Messages* uses the aircraft dynamics data, the grid winds data, and the estimated ground times to predict when each event along the flight path will occur.

In addition to performing updates of the databases, *Process Flight Messages* also generates flight update messages for the *Flight Table Manager*, event list update messages for the *Traffic Demands Database Processor*, and FA messages for the *Parser*. It also maintains two supporting data structures called *time out arrays*. These arrays are used by *Remove Late Departures From TDB* and *Time Out Flights From FDB* to facilitate their respective processing.

Remove Late Departures From TDB sends delete flight transactions for late departing flights to the *Traffic Demands Database Processor* (*TDB*). This module receives the current time from NAS messages and accesses the Departure Time Out Array in order to determine which flights need to be deleted from the TDB. As an adjunct to this processing, *Remove Late Departures From TDB* also generates a time transaction, which contains the current NAS message time, for the TDB.

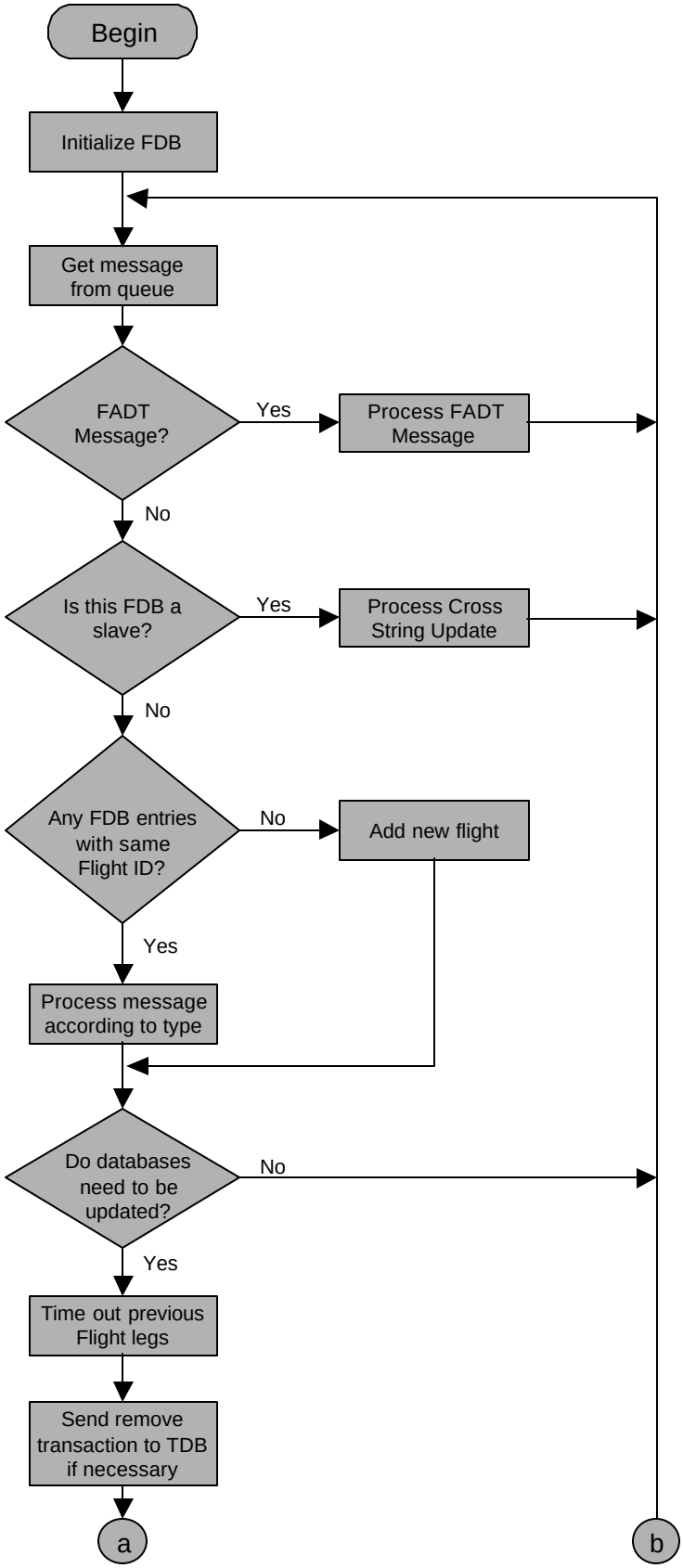

Figure 25-12. Sequential Logic for the FDB_Manager Process

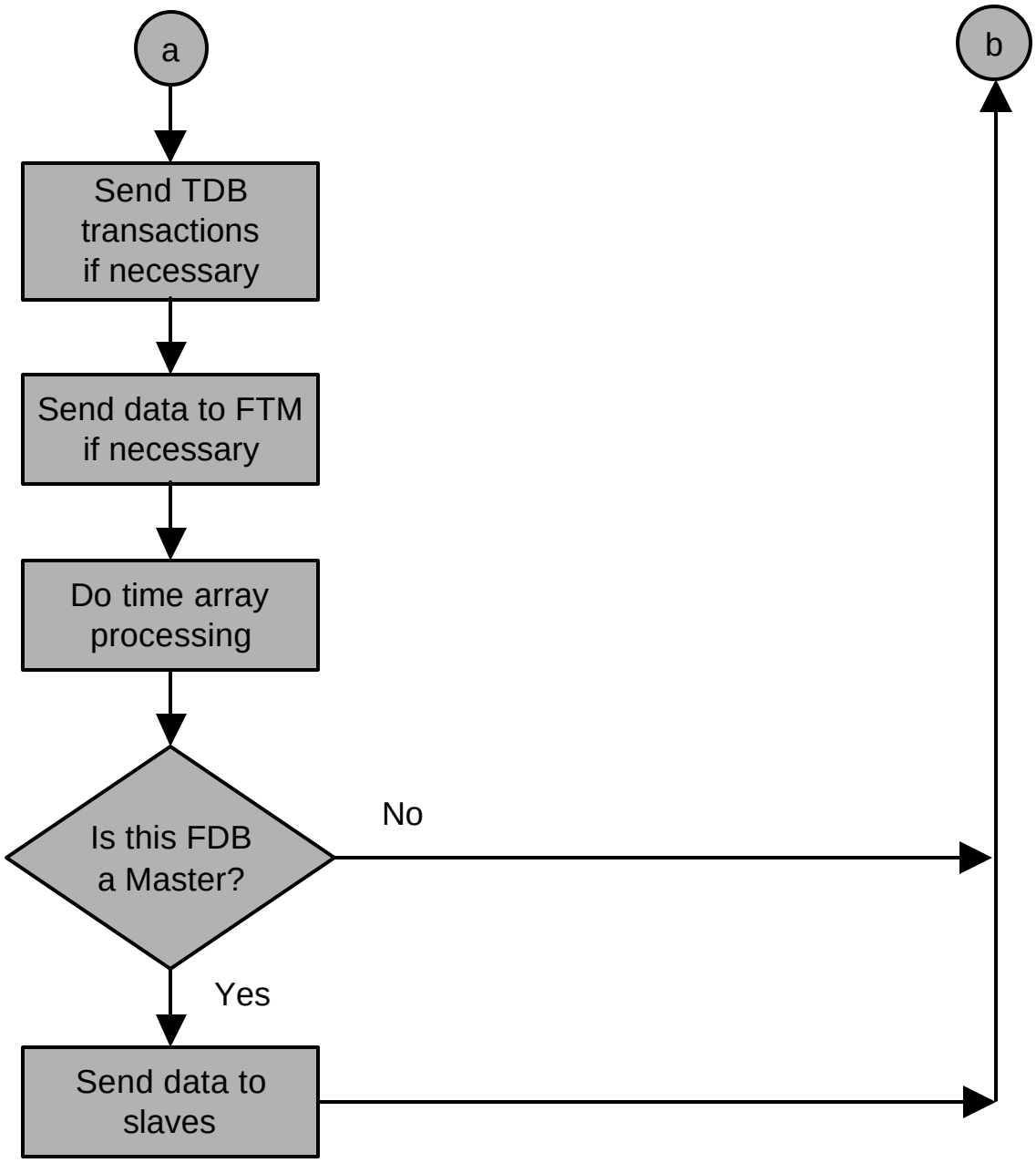

Figure 25-12. Sequential Logic for the FDB_Manager Process (continued)

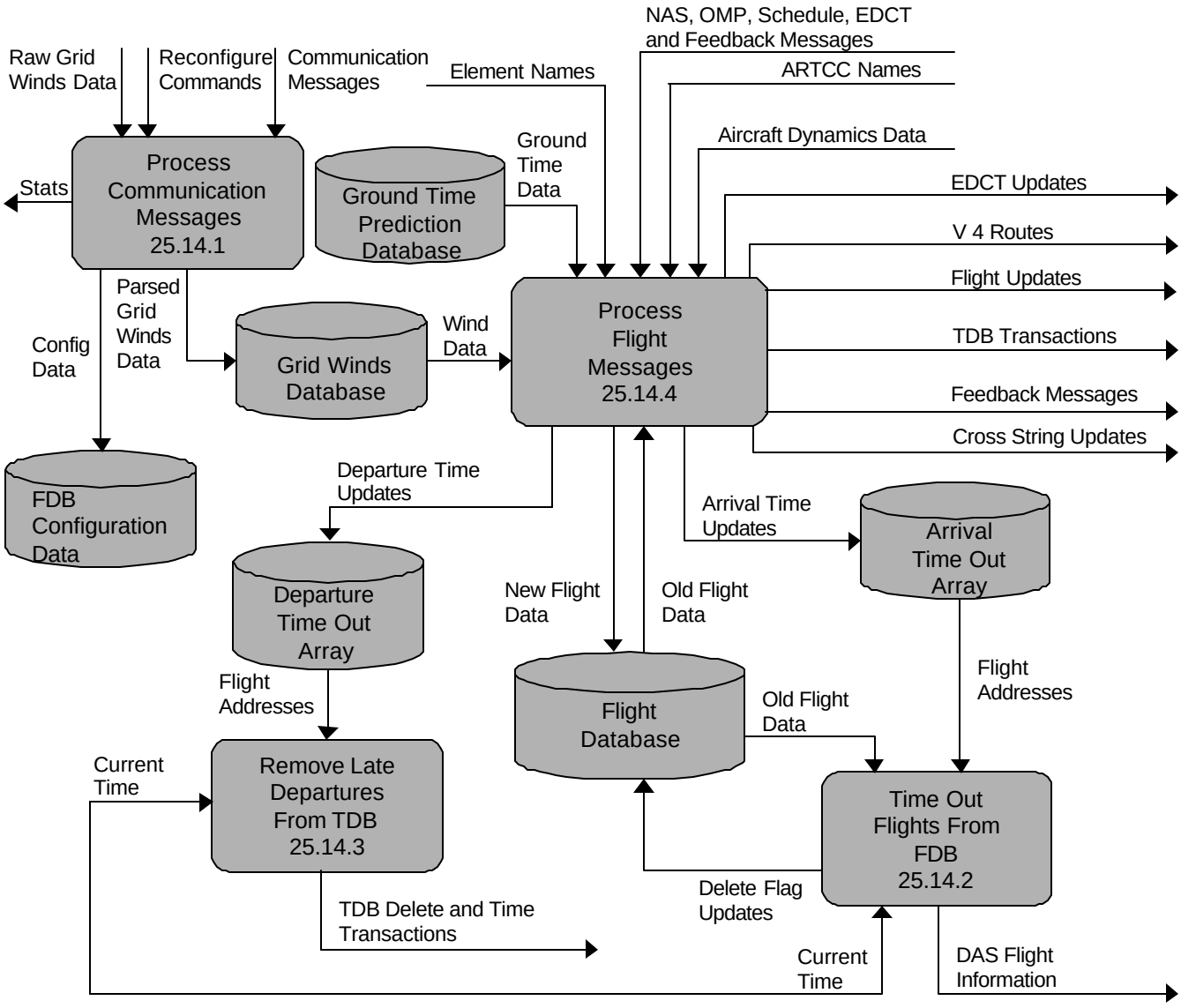

Figure 25-13. Data Flow of the FDB Manager Process

Time Out Flights From FDB also receives the current NAS message time as input. The task of this module is to determine which flights in the Flight Database are no longer in use. For this purpose, *Time Out Flights From FDB* accesses the Arrival Time Out Arrays in order to determine which flights should be deleted from the database. As flights are deleted, *Time Out Flights From FDB* generates flight information messages, which are sent to the *Ground Time Prediction Server*.

The module *gridwinds_read* is responsible for updating the grid winds database with winds aloft data contained in raw **grid winds data files**. This module is triggered by messages from the external weather server. When triggered, *gridwinds_read* parses the raw grid winds data file and updates the grid winds database. The winds data is used by *Process Flight Messages* in order to correct each flight's filed air speeds for the effects of winds.

25.14.1 The Process Communication Messages Module

Purpose

The *Process Communication Messages* module is responsible for handling certain communication events. These events include

- Notification that flight messages or grid winds updates are available
- Reconfigure or statistics requests are pending
- Communication timeout indicators
- Indicators of problems with child processes

The *Process Communication Messages* module determines which type of event has occurred and what type of processing is necessary to handle it.

Input

From the node switch port, *Process Communication Messages* receives the grid winds update input file name, reconfigure requests, and statistics requests. From the receive queue, it receives flight messages from the Parser, and EDCT.

Output

Process Communication Messages may return a flight message to the calling routine or a statistics reply to the requestor.

Processing Overview

The *Process Communication Messages* module consists of three sub-modules: *GetNextMessage, gridwinds_read,* and *comserver_timeout*. Figure 25-14 illustrates the data flow among these three modules.

The *GetNextMessage* module (see section 25.14.1.1) waits for a communication event to occur. Depending on the event type, processing is as follows:

- **node-switch event** –– This indicates that one or more messages are available from the node switch. *GetNextMessage* reads each message from the port, and checks to see whether any of them are grid winds update messages. (These are indicated by the word "aloft" at the beginning of the message, followed by the name of the grid winds update file.) All grid winds update messages are processed by module *gridwinds_read* (see section 25.14.1.2). Reconfigure and statistics requests are processed by *GetNextMessage* with statistics responses sent back to the requestor. All other types of messages in the port are ignored. This continues until the port is empty.
- **timeout event** –– This type of event occurs when the node switch connection has been severed, or when no node switch messages have been received within a

certain time interval. These events are handled by the *comserver_timeout* module (see section 25.14.1.3).

- **receive-queue event** –– This type of event indicates that a flight message from the Parser or EDCT Processor is available in the receive queue. *GetNextMessage* attempts to dequeue the message. If more than 0 bytes are retrieved, then it is assumed that a flight message has been successfully received. This message is returned to the calling routine.
- **child-process-crashed event** –– This type of event indicates that a child process has died. *GetNextMessage* determines which process it was, and restarts it. If the process has died and been restarted a great many times, a warning is displayed for the operator.

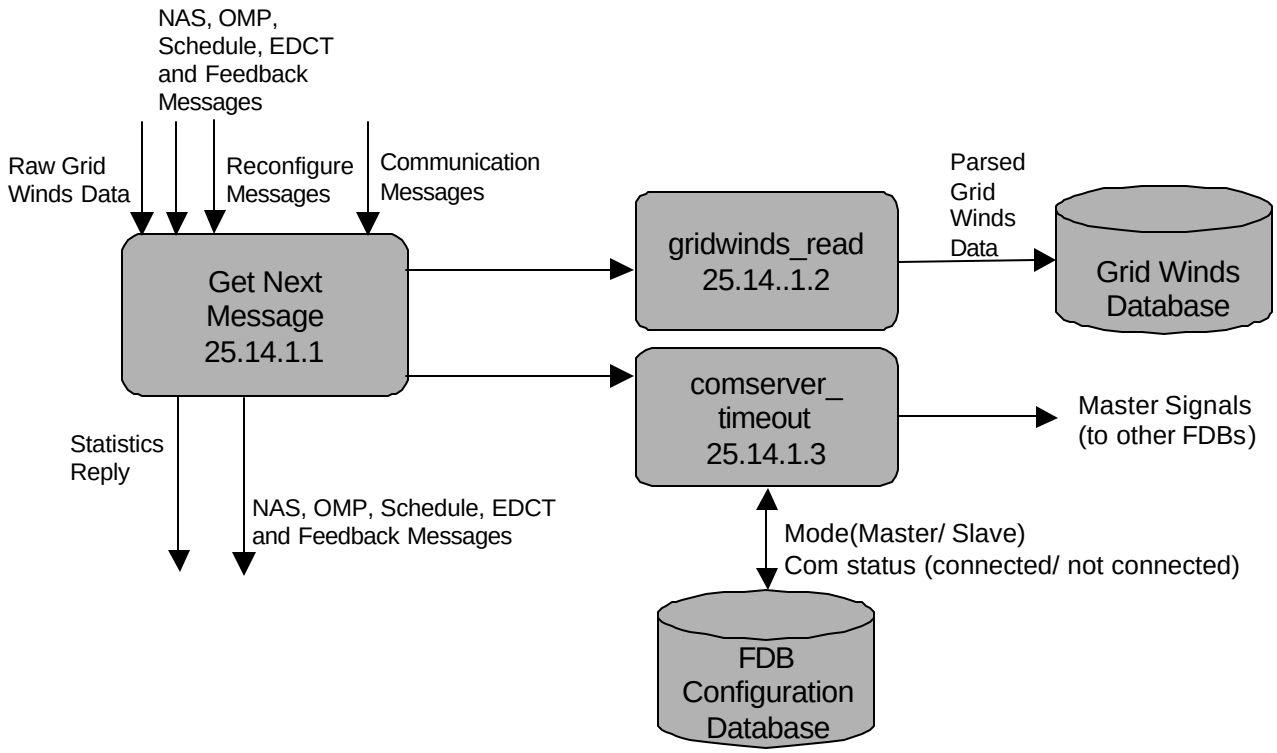

Figure 25-14. Data Flow of the Process Communication Messages Module

25.14.1.1 The GetNextMessage Module

Purpose

The *GetNextMessage* module determines which type of communication event has occurred and what type of processing is necessary to handle it.

Input

GetNextMessage receives an indication of the type of event that has occurred. From the node switch port, it receives the grid winds update input file name as well as reconfigure commands and statistics requests. From the receive queue, it receives flight messages from the Parser and EDCT Processor.

Output

GetNextMessage returns a flight message to the calling routine or a statistics reply to the requestor.

Processing

Processing depends on the event type.

If it is a node-switch event, it is an indication that messages are available in the node switch port. *GetNextMessage* reads each message from the port; messages which include grid winds input file names are passed to the *gridwinds_read* module and the grid winds database is updated. Reconfigure commands can be used to update the database. For statistics requests, *GetNextMessage* produces a reply and sends it to the requestor. Any other type of message in the node switch port is ignored. All available messages in the port are processed before checking for other events.

If it is a timeout event, it is an indication that the node switch connection has been severed, or that no network addressing messages have been received within a certain time interval. The *comserver_timeout* module is invoked to handle the timeout processing.

If it is a receive-queue event, it is indication that a flight message from the Parser or EDCT processor is available in the receive queue. *GetNextMessage* attempts to dequeue the message. If more than 0 bytes are retrieved, then it is assumed that a flight message has been successfully received. This message is returned to the calling routine.

If it is none of the above type of events and if the event type is within the defined set of recognized event types, then it is an indication that a child process has died. *GetNextMessage* determines which process it was, and restarts it. If the process has died and been restarted a great many times, a warning is displayed for the operator.

GetNextMessage repeats this processing until it has successfully received a flight message.

See Figure 25-15 for a sequential logic diagram for this module.

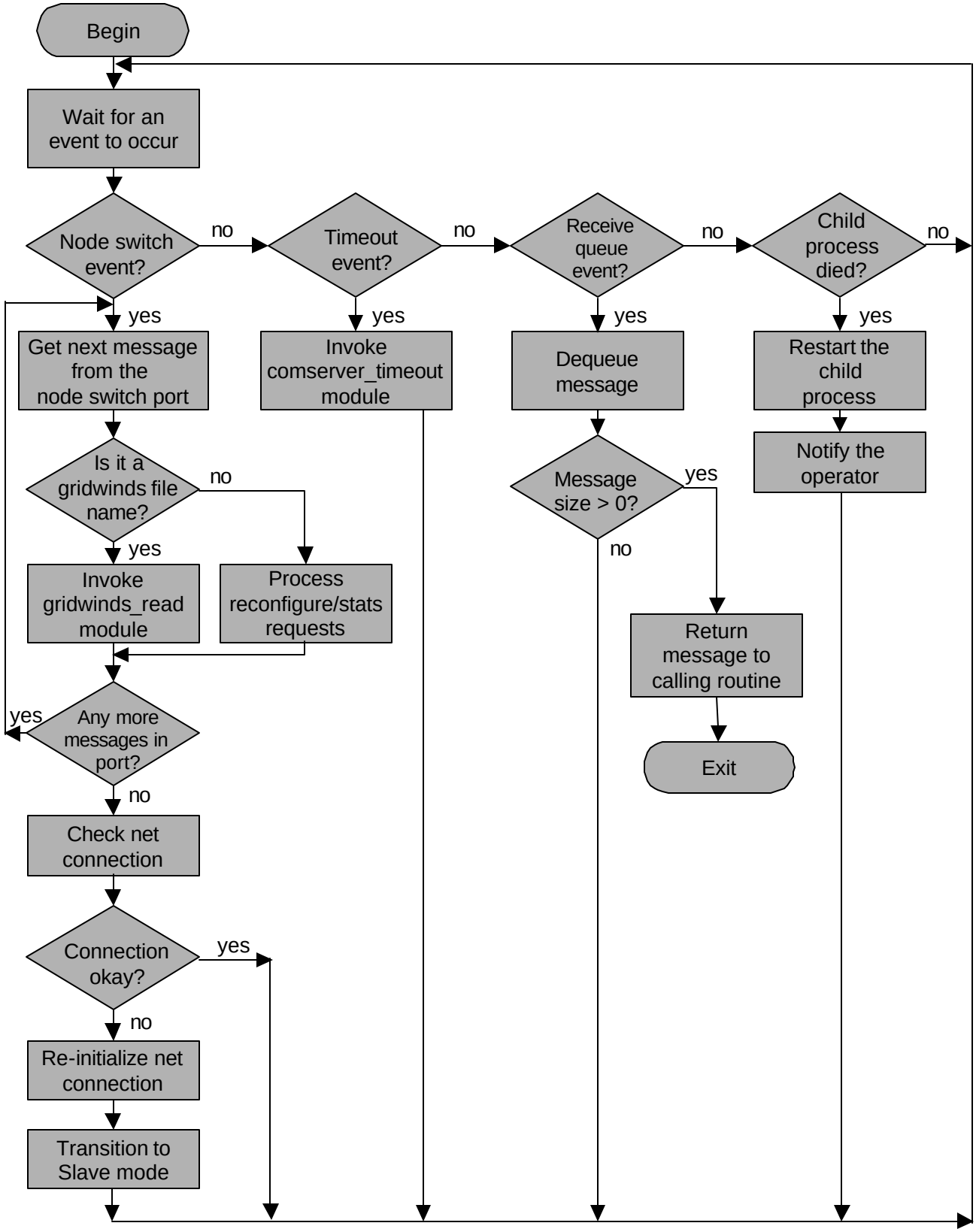

Figure 25-15. Sequential Logic for the GetNextMessage Module

25.14.1.2 The gridwinds_read Module

Purpose

The *gridwinds_read* module is responsible for obtaining data from grid winds files and storing that data into the **grid winds database** that is accessed by *Process Flight Messages*.

Design Issue: The Grid Winds Database

The grid winds database contains time-indexed winds data for fifteen altitude levels across the greater continental United States. It is implemented as a mapped file, **gwdb**, consisting of two distinct sections. The first section includes a control array of time records, a latitude array, a longitude array, and various control fields. The control array acts as the time interface to the winds data. The latitude array and the longitude array provide quick access to the boundary points of the grid cells. Tables 25-1 and 25-2 provide detailed descriptions of the data structures describing the first section of the mapped file. The second section, the data section, contains a time-indexed array of winds data sets. Each data set is organized as a 3-dimensional array indexed by 15 altitudes, 62 latitudes, and 81 longitudes. Each array element contains wind direction and wind speed. Tables 25-3 through 25-5 provide detailed descriptions of the data structures describing the second section of the mapped file.

The winds grid is based on a polar stereographic projection of the greater continental United States. This projection has the characteristic that grid cell boundaries do not fall along lines of equal latitude or longitude. This fact makes the problem of finding the grid cell corresponding to a given latitude and longitude quite difficult. Because this type of lat/lon query is the only means of access to the winds data, the search must be made simple and quick. This is accomplished in the following manner.

The polar stereographic grid cell boundaries are converted to a Cartesian coordinate system using the following conversion formulas:

r = cot(45+lat/2)
x = r*sin(lon-lon0)
y = -r*cos(lon-lon0) where lon0 is -105.

The resulting (x, y) pair represents the converted (lon, lat) pair. All cells in a given column of the winds grid have identical x values and all cells in a given row of the wind grid have identical **y** values. During initialization, the FDB reads a file containing winds grid lat/lon intersection points, converts these points to **x**/**y** values, and stores the series of **x** values in the gwdb longitude array and the series of **y** values in the gwdb latitude array. These steps are illustrated in Figure 25-16.

A typical access to the winds database is accomplished by using the control array and input time to determine the proper data set time index, converting a given (lon, lat) pair to its corresponding (**x**, **y**) pair, searching the latitude array for the index corresponding to the **x** value, searching the longitude array for the index corresponding to the **y** value, and using these indices along with an altitude index to read the proper winds record.

Table 25-1. Gw_header_t Data Structure

gw_header_t				
Library Name: /atms/libraries/fdb_openlib	**Purpose:** This structure defines the first of two distinct sections in the grid winds database. It is the header section which contains x/y coordinate data as well as the time control array.			
Element Name: gridwinds.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
control	Control array containing time data	Array sorted by start times	1..gw_max_periods	array of gw_control_rec_t
default	Control structure containing indices to most recent winds data	Single record		gw_control_rec_t
y	Array of latitude indicies	Array of 62 converted latitudes		array of float
x	Array of longitude indicies	Array of 81 converted longitudes		array of float
lation_loaded	Flag indicating whether y,x arrays successfully initialized		T, F	Boolean
winds_loaded	Flag indicating whether winds database contains any data		T, F	boolean

Table 25-2. Gw_control_rec_t Data Structure

Gw_control_rec_t				
Library Name: /atms/libraries/fdb_openlib	**Purpose:** This structure describes a control record which maintains time data. The period index field is the time array index into the second section of the grid winds database.			
Element Name: gridwinds.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
time_update	The time that the file containing this wind data was processed.	System clock format		CALTIME
start_clk	The beginning of the time span for which this data is valid.	System clock format	rounded to nearest hour	CALTIME
end_clk	The end of the time span for which this database is valid.	System clock format	rounded to nearest hour	CALTIME
exact_clk	The center of the time span for which this data is valid.	System clock format	rounded to nearest half hour	CALTIME
start_mam	The value of strat_clk above in minutes.	Minutes elapsed since Jan,1, 1980		INT32
end_mam	The value of end_clk above in minutes.	Minutes elapsed since Jan.1, 1980		INT32
exact_mam	The value of excat_clk above in minutes.	Minutes elapsed since Jan 1. 1980		INT32
period_index	The time index into winds data array related to this record.	Time array index	1.gw_max_periods	short
valid_data	Is the winds data for this time period valid?		T, F	boolean
input_file	Name of the file that provided winds data for this record.	Ascii string		string80

Table 25-3. Gw_header_t Data Structure

Gw_period_array_t				
Library Name: /atms/libraries/fdb_openlib		**Purpose:** This structure defines the second section of the grid winds database. This is the section that contains the actual winds data broken up by time.		
Element Name: gridwinds.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
gw_period_array_t	Array of winds data sets.	array	1.gw_max_periods	array of gw_period_t

Table 25-4. Gw_period_t Data Structure

Gw_period_t				
Library Name: /atms/libraries/fdb_openlib		**Purpose:** This structure contains a full set of winds data for a given time range		
Element Name: gridwinds.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
gw_period_t	3-deminsional array defining a set of winds data	15 altitutdes. 62 latitudes, 81 lons	15x62x81	array of gw_item_rec_t

Table 25-5. Gw_item_t Data Structure

Gw_item_t				
Library Name: /atms/libraries/fdb_openlib		**Purpose:** This structure contains wind direction ans wind speed for a single grid winds cell.		
Element Name: gridwinds.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
wind_direction	Grid wind direction measured in “to” redians.W->E=90deg.	radians	0 – 6.28	float
wind_speed	Grid wind speed	knots	>=0	short

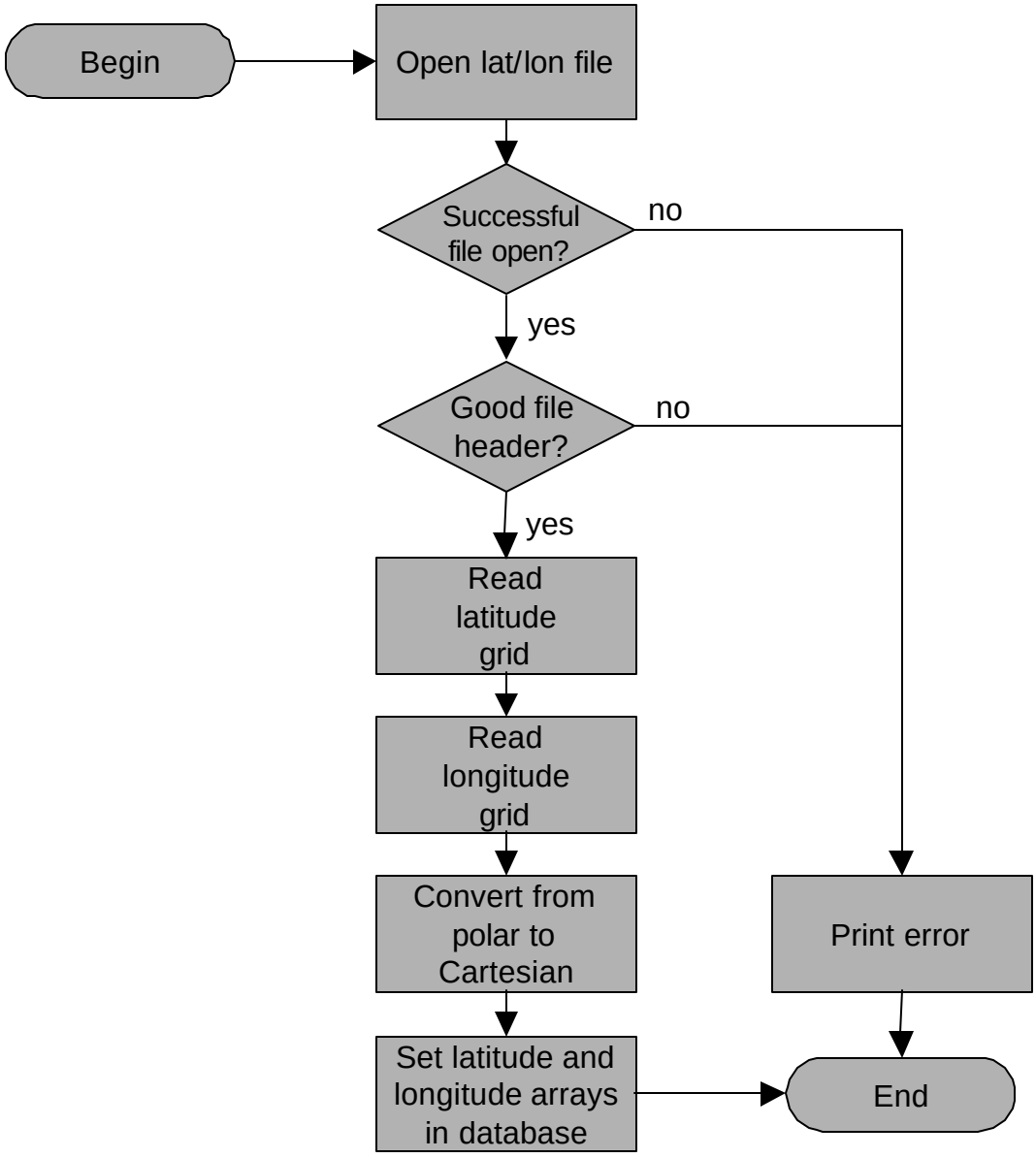

Figure 25-16. Sequential Logic for the gridwinds_latlon Routine

Input

Gridwinds_read receives two types of input

(1) Messages from the communications interface containing a grid winds filename. Receipt, normally occurring at 3-hour intervals, indicates that a new raw grid winds file is available. The message contains the filename.

(2) The raw grid winds file itself. The file contains winds data separated into 30 unit grids. There are fifteen sets of grids. Each set corresponds to a 3000-foot altitude range and contains a direction grid and a speed grid. Each grid contains 81 X 62 grid cells.

Output

The *gridwinds_read* module stores wind direction and speed in the grid winds database used by *Process Flight Messages*. The grid winds database structure is shown in Tables 25-1 through 25-5.

Processing

Gridwinds_read receives a gridwinds filename from the communications interface. *Gridwinds_read* parses the message, extracting the gridwinds filename. The routine opens the file, verifies its size, and reads the winds data into temporary storage grid by grid. It reads a direction grid, then a speed grid for a single altitude level. It repeats the same process until reaching 15 altitude levels.

Range checking is performed on the wind speed and direction data. A value of 99999 for wind speed indicates that the file has no data for that particular wind cell. In such cases, *gridwinds_read* stores a zero for the wind speed. If all grids are successfully read and verified, *Gridwinds_read* converts the input filename into a time index and transfers the new winds data from temporary storage to the appropriate location in the grid winds database. The processing steps performed by *Gridwinds_read* are depicted in Figure 25-17.

Error Conditions and Handling

When *gridwinds_read* encounters an error in processing a winds file, it writes an error message to the screen, and discards the entire winds file. The grid winds database remains unchanged. The following is a list of errors:

(1) Unable to open grid winds file.

(2) Incorrect file size or header data.

(3) Direction or speed data out of bounds. Valid directions are 0-359 degrees (these converted to radians for internal storage). Valid speeds are greater than zero.

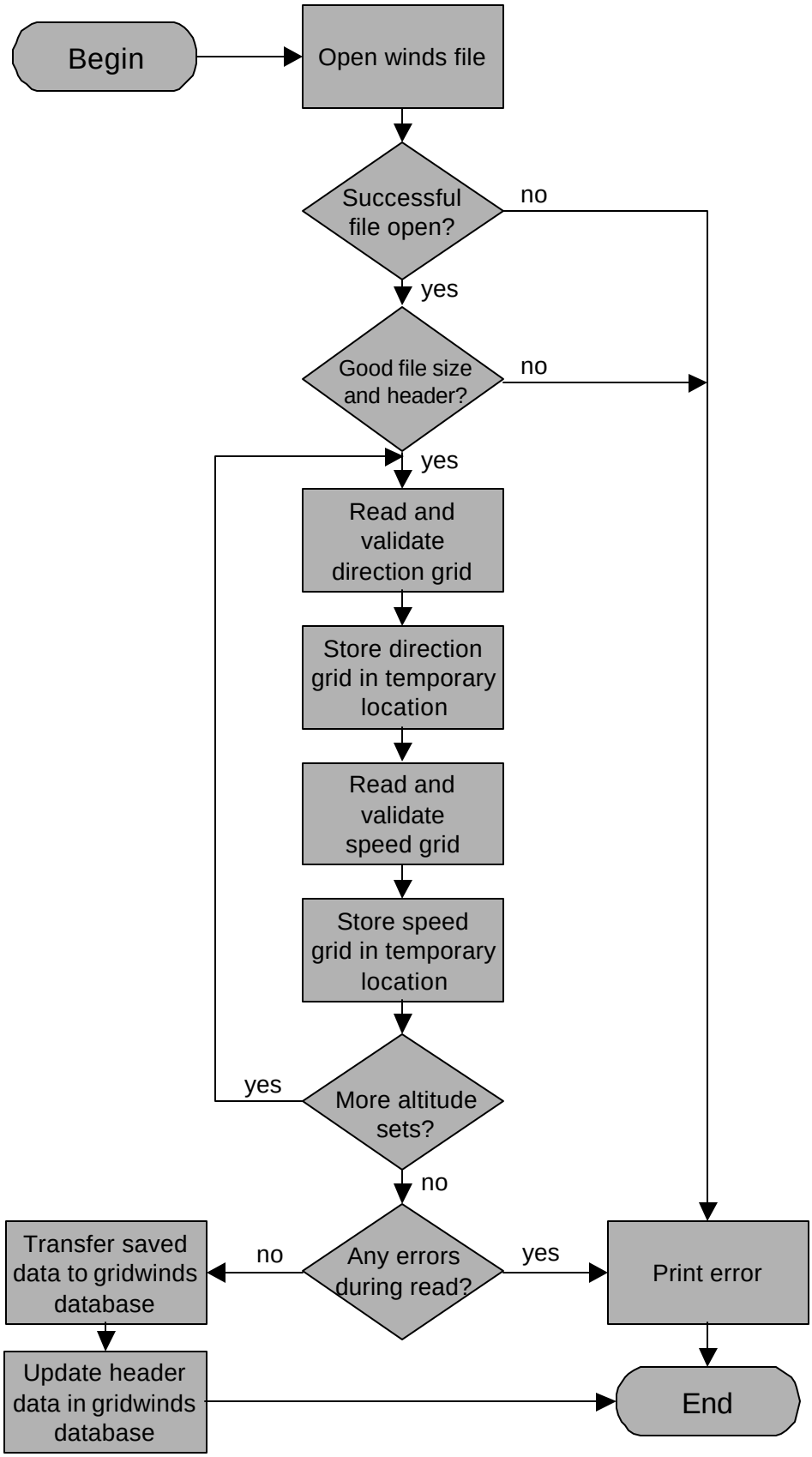

Figure 25-17. Sequential Logic for the gridwinds_read Routine

25.14.1.3 The comserver_timeout Module

Purpose

The purpose of the *comserver_timeout* module is to determine whether the FDB should transition from Master to Slave mode, or vice-versa.

Input

This module accesses the FDB global variables that indicate whether this FDB is in Master or Slave mode.

Output

This module updates the FDB global variable that indicates whether this FDB is in Master or Slave mode.

Processing

If the connection between the FDB and the node switch has been lost, then an attempt is made to reconnect; otherwise processing depends on whether the FDB is in Master or Slave mode.

If the FDB is in Master mode when the timeout occurs, this module checks to see whether there have been more than nine FDB Master pulse signals since the last TZ message was received from the Parser. If so, the FDB transitions to Slave mode. If FDB did not decide to switch to Slave mode, then it sends Master pulses to all other FDB Slave managers, as well as to the FDB2 on this string, and it also sends an AF message to the Parser to tell it to continue sending data. If FDB was not able to send the Master pulses, then the FDB transitions to Slave mode.

If the FDB is in Slave mode when the timeout occurs, it attempts to verify that its net connection is still valid. If it is not, then it attempts to reconnect. Otherwise, if cross-string messages are still being received from the Master FDB, then the FDB stays in Slave mode but prepares to transition to Master mode by notifying the Parser to start sending it data. If it is not receiving data, then it transitions to Master mode; it notifies the Parser to start sending it data and it sends out the Master pulses to the other FDBs.

After these checks have been performed, if the FDB's connection to the node switch is down, it attempts to reconnect.

At the end of timeout processing, this module resets the timeout counter to one of two user-defined values, depending on whether the FDB is in Master or Slave mode, and whether the net connection is up or down.

See figure 25-18 for a logic diagram for this module.

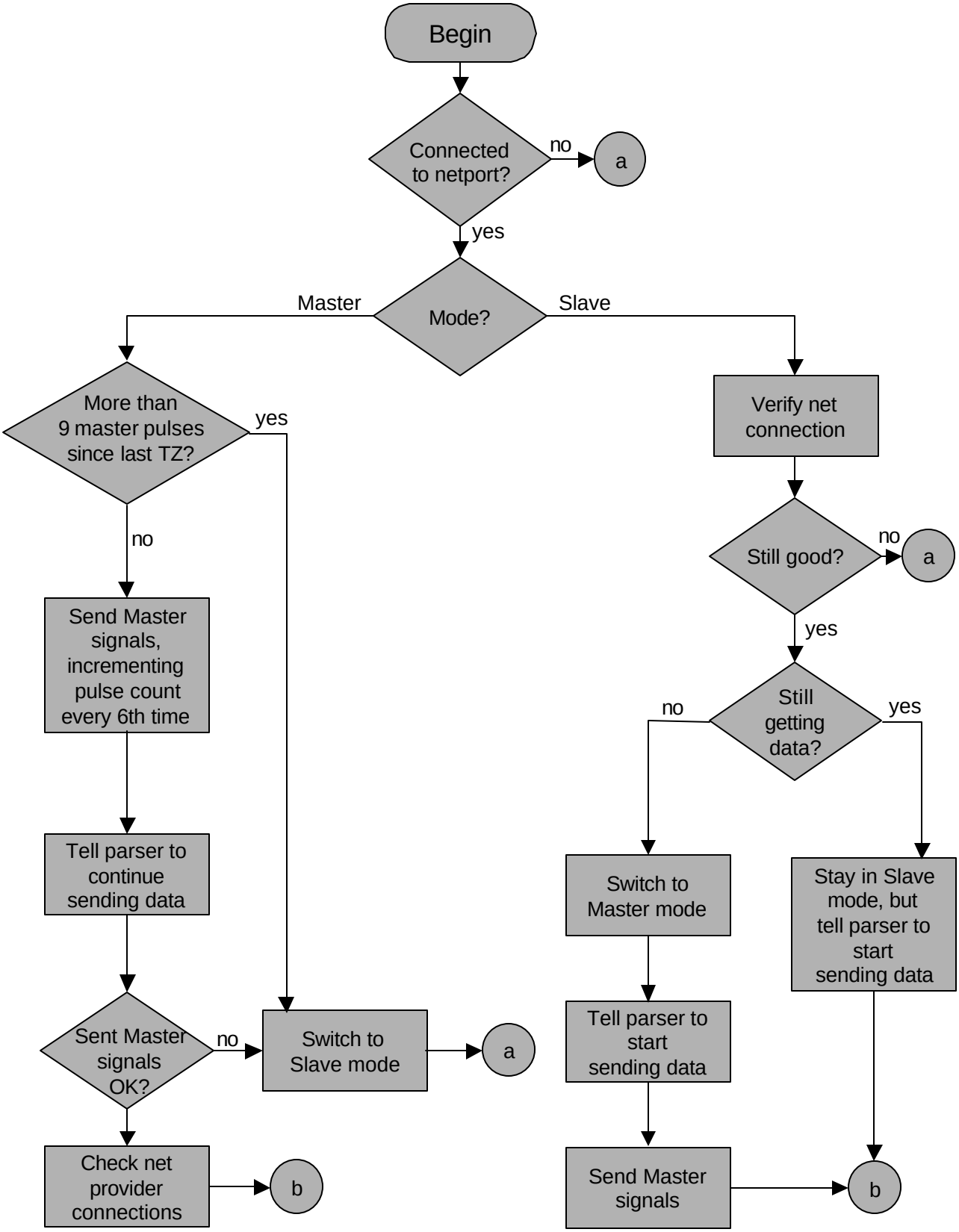

Figure 25-18. Sequential Logic for the comserver_timeout Module

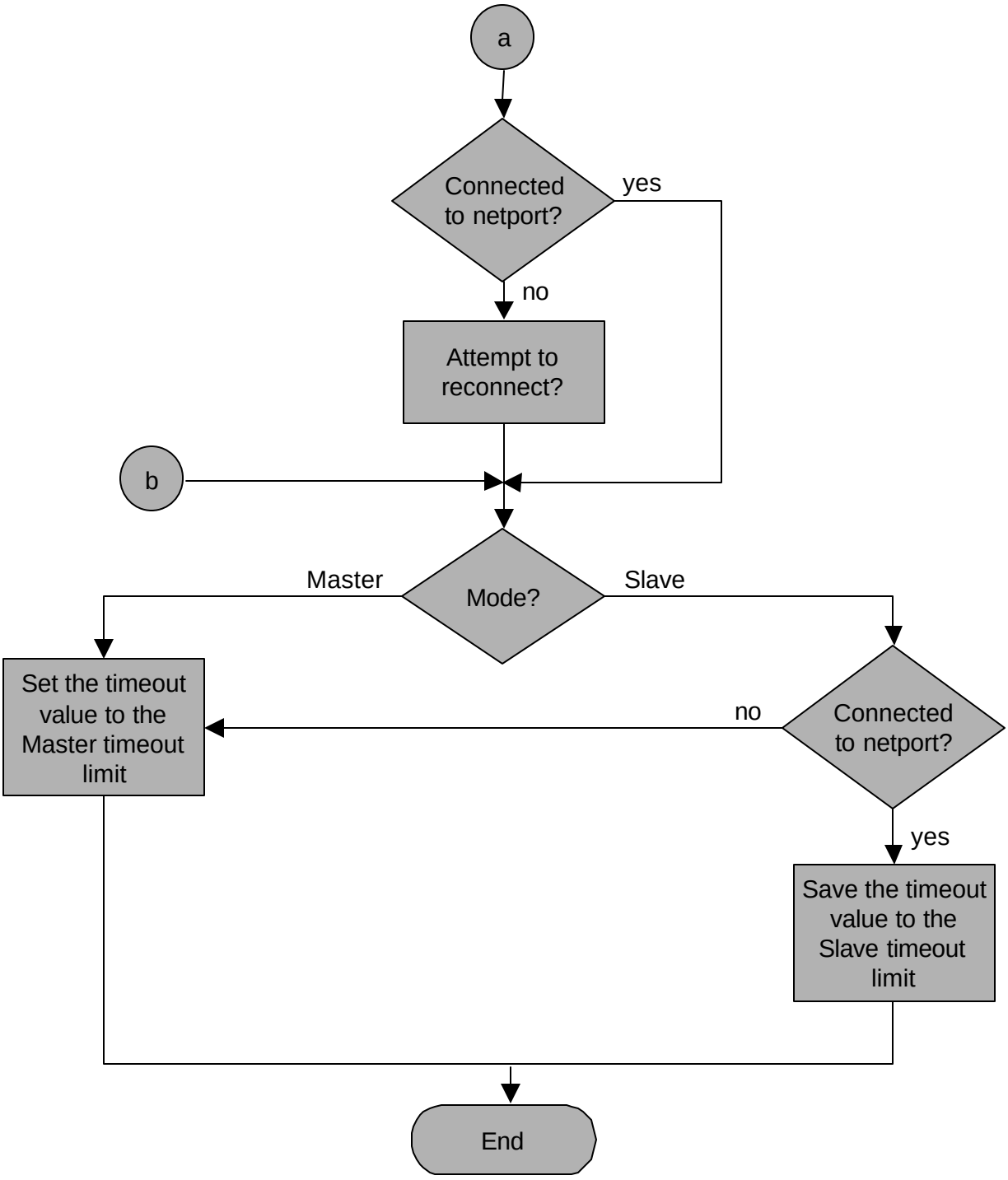

Figure 25-18. Sequential Logic for the comserver_timeout Module (continued)

25.14.2 The Time Out Flights From FDB Module

Purpose

The *fdb_manager* process is designed to run *ad infinitum*, maintaining a database of all current and proposed flights in the NAS. Since this process runs on a computer that has some finite amount of memory, the design must include some way to reuse memory.

The module *Time Out Flights From FDB* addresses this requirement; it frees up memory by deallocating slots, each of which contains information about one flight. *Time Out Flights From FDB* reallocates a slot by marking the record as available for reuse by *Process Flight Messages*. It deallocates a slot when it determines that no further messages will be received affecting the flight in that slot. This determination is made by comparing the arrival time of the flight with the current time.

Design Issue: the *T*ime Out Array

The manner in which *Time Out Flights From FDB* uses the flight's status and arrival times to schedule the flight's slot for deallocation is illustrated in Figure 25-19. **Time-out arrays** hold linked lists of addresses of flight slots in time order such that each linked list is associated with a particular time interval. Arrays (one for each flight status) are used to group flights into 15-minute intervals by status and arrival time. The arrival time is converted into a time interval number. This number is used as the subscript of the array element in which a linked list resides. The linked list includes the address of the flight slot.

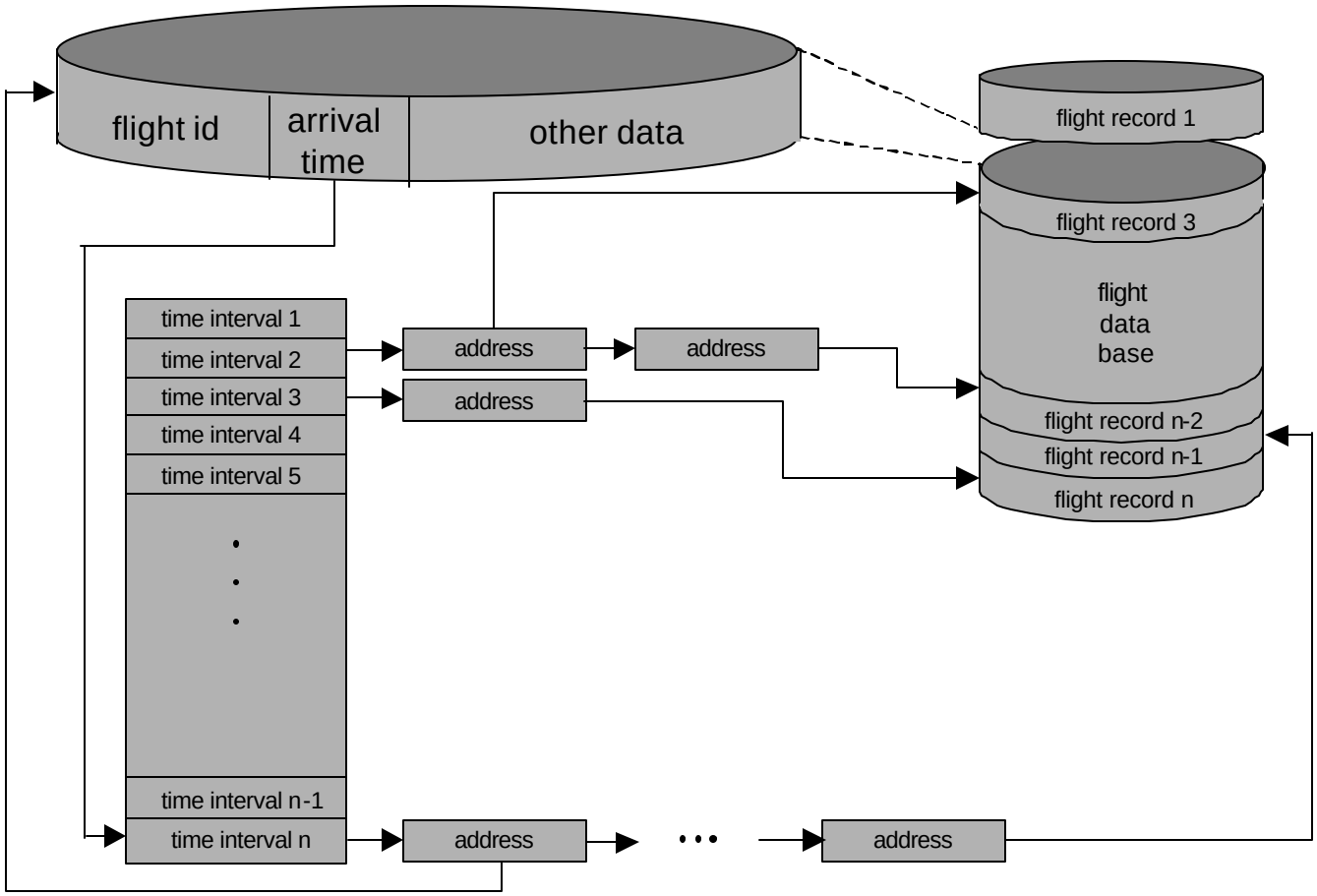

Figure 25-19. A Time Out Array Relates the Arrival Time to a Flight's Address

Input

Time Out Flights From FDB receives the following input:

(1) Current time as determined from the time stamps of incoming NAS messages.

(2) Old flight data from the flight database.

(3) Flight addresses from the arrival time-out arrays.

Output

Time Out Flights From FDB creates as output:

(1) Flight header records and event data blocks which have been marked as deleted.

(2) DAS flight message transactions for each flight that is deleted.

Processing Overview

*Time Out Flights From FDB (*shown in Figure 25-20) uses the time out arrays described in the previous section to decide when flight data is no longer needed. *Time Out Flights From FDB* removes flights from the main databases (the FDB and EVDB) and from the supporting data structures (time out arrays and hash table) during *Delete Old Flights* processing. *Delete Old Flights* then supplies the addresses of flights to be deleted to *Generate DAS Flight Messages*. This module generates a deleted flight information transaction for the Ground Time Prediction server.

25.14.2.1 The Delete Old Flights Module

Purpose

The purpose of *Delete Old Flights* is to determine a time interval range for which flights should be deleted and then delete those flight records whose arrival times fall within that range. This module also passes deleted flight addresses to *Generate DAS Flight Messages* and, if enabled, will also save deleted flight information to a disk file.

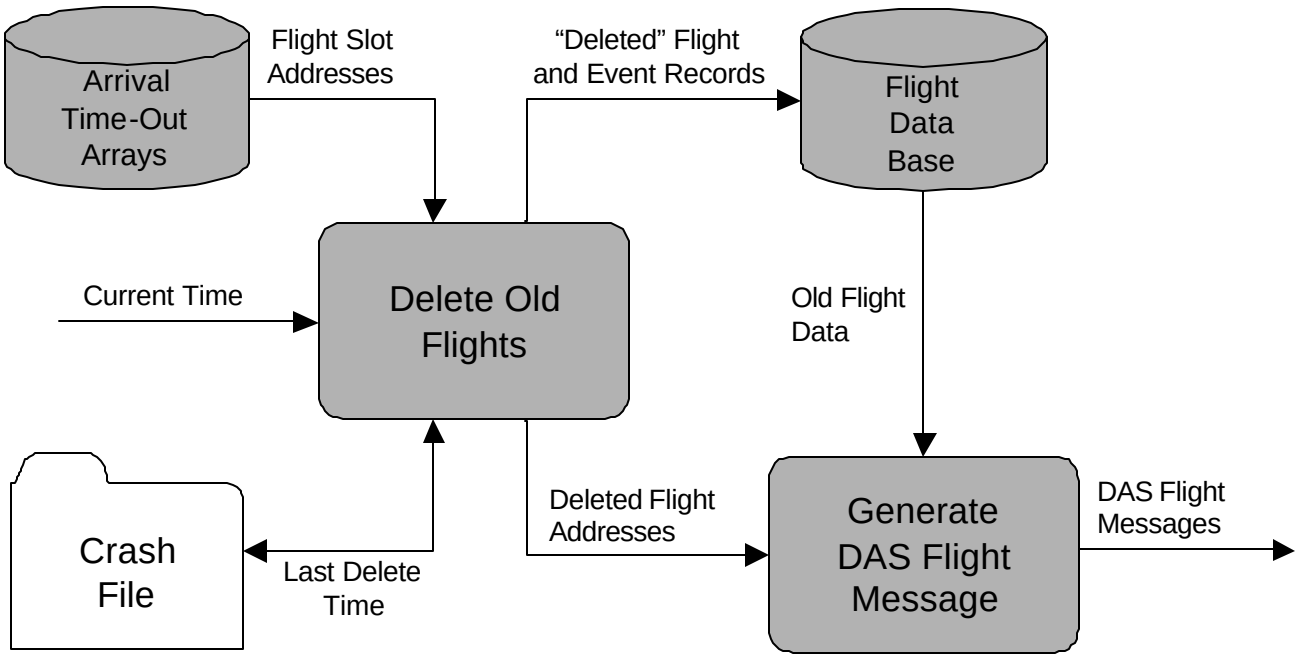

Figure 25-20. Data Flow of the Time Out Flights From FDB Module

Input

Delete Old Flights receives as input:

(1) Flight slot addresses – addresses (offsets into the map file) from the **time-out arrays** for all flights to be deleted

(2) Current time – from the last NAS message received

(3) Time when flights were last timed out – this time was saved when flights were last timed out.

Output

Delete Old Flights produces the following output:

(1) Updates to the FDB and EVDB – mark the flight record header and flight event blocks as deleted; delete flight entry from the hash table and arrival **time-out array**.

(2) Time when flights were last timed out – update the previous value in the crash file upon completion of *Delete Old Flights.*

Processing

Delete Old Flights first calculates the time interval number associated with the current time. It then compares this interval number with the interval number when flights were last timed out. If the current time interval is greater than the last time-out interval, *Delete Old Flights* then calculates a range of time intervals for which flights should be timed out. This interval range goes from two hours before the last time-out interval until one hour before the current interval. After calculating the interval range, *Delete Old Flights* traverses each of the time-out arrays, performing a number of steps for each flight.

If the *fdb_manager* process was invoked with the **-d** switch (see Section 25.14 for a complete description of *fdb_manager* process execution control), *Delete Old Flights* first sends deleted flight information to a disk file. *Delete Old Flights* then passes each flight record address to *Generate DAS Flight Messages* (see Section 25.14.2.2), which generates a deleted flight transaction for the Ground Time Prediction server. Next, *Delete Old Flights* updates the **fdb**, **evdb** and all supporting data structures through the following steps:

(1) Delete each of the flight's event blocks by setting the **distance** field in the first event for each block to **-1**.

(2) Remove the flight's entry from the hash table.

(3) Remove the flight's entry from the time array.

(4) Delete the flight record header by setting its **deleted** field to TRUE.

After processing each of the flights in the calculated time-out array interval range in this manner, *Delete Old Flights* records the current time in the crash file as the time when flights were last timed out.

Error Conditions and Handling

During the course of processing a given flight, *Delete Old Flights* could encounter corrupted data in the **fdb**, **evdb**, the hash table or the time-out arrays themselves. Depending on which of these structures is corrupted, and in what manner, *Delete Old Flights* takes different courses of action.

No provision is made within the code for checking legal address bounds. The severity of these types of errors will be fatal; a fault traceback for the current point of execution will be saved in the main *fdb_manager* transcript pad, and the program will terminate.

Lower severity errors include redundant entries for a flight in the time-out arrays or inclusion of previously deleted flights within that data structure. In each case, an error message will appear in the main *fdb_manager* transcript pad indicating the nature and location of the error. If applicable, information on the offending flight will also be displayed. Errors which indicate corruption of either the **fdb** or **evdb** will result in termination of *Delete Old Flights* processing. Processing will continue for errors detected in the supporting data structures.

25.14.2.2 The Generate DAS Flight Messages Module

Purpose

Generate DAS Flight Messages prepares flight information and route event data to be sent to the *Ground Time Prediction System*.

Input

The input for *Generating DAS Flight Messages* is a flight address. Depending on a flight status, the *Process Flight Data* determines which array to be used to get a flight address.

Output

The output of the *Generate DAS Flight Messages* module is a DAS Flight Message. This DAS Flight Message consists of message type, flight record, and event lists. The **fdb_das interface** data structure, which is used for this message, is shown in Table 25-6.

Table 25-6. fdb_das_interface_t Data Structure

fdb_to_das_t				
Library Name: /atms/libraries/gtp_openlib		**Purpose:** To store information to be shipped from the FDB to the *Ground Time Prediction*		
Element Name: fdb.das.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
flight _record	The record containing all information about a flight.	A C structure		flight_db_type
event_list	The path of this flight in terms of elements it crosses.	A C structure		ev_list_array_t

Processing

When a flight is timed out from the *Flight Database, Generate DAS Flight Messages* uses the flight record address to extract flight record and event list information for each flight. *Generate DAS Flight Messages* packs the flight information and sends it, along with event list data, to the *DAS Relay*, which forwards each DAS Flight Message to the *Ground Time Prediction System*. If there are no events for the flight, the *Generate DAS Flight Messages* does not generate any information.

Error Conditions and Handling

Errors cause an error message to appear in the main process window of the ETMS system operator's node.

25.14.3 The Remove Late Departures From TDB Module

Purpose

Remove Late Departures From TDB monitors scheduled and proposed flights for delayed departures. This is done so that flights which are very late in departing do not affect traffic demands in the TDB. Information about such flights is removed from the TDB approximately five minutes after the scheduled/proposed departure time. The flight state and route information is retained in the FDB, however. If another message is received updating the status of that flight, its route information is added once again to the TDB. If no such message is received, the flight will eventually be removed from the FDB as described in Section 25.14.2.1.

Input

Remove Late Departures From TDB receives as input:

(1) Current time –– This is the time stamp from the last NAS message received.

(2) Flight addresses –– Obtained from the departure time-out array (identical to the arrival time-out arrays described in Section 25.14 except that the array subscript is related to departure time); used to access flight departure times and determine which flights need to be removed from the TDB.

(3) Flight data –– Obtained from the **fdb** and **evdb**, used to generate a delete transaction for those flights which are to be removed from the TDB.

Output

Remove Late Departures From TDB generates as output:

(1) TDB transactions –– Delete transactions for flights whose departure has been delayed and a time transaction that contains the current time.

(2) FDB record updates –– Each FDB record that is deleted from the TDB is marked to indicate the lack of a TDB entry for that flight.

Processing

Remove Late Departures From TDB compares the current time to the time when flights were last timed-out from the TDB. If at least one minute has elapsed, *Remove Late Departures From TDB* then calculates a time interval range for which flights should be timed out. This interval range starts one interval before the last time-out interval and ends with the current interval. *Remove Late Departures From TDB* then traverses the departure time-out array and times out any flights whose departure is at least five minutes before the current time. As candidate flights are located, *Remove Late Departures From TDB* generates a delete transaction buffer and enqueues that buffer for transmission to the TDB by the *TDB Relay*. Finally, *Remove Late Departures From TDB* generates a time transaction buffer containing the current time and enqueues that buffer for transmission to the TDB.

Similar to the removal of flights from the FDB, if the **-d** switch was enabled when the *fdb_manager* process was started, *Remove Late Departures From TDB* will send deleted flight information to a disk file.

Error Conditions and Handling

When a flight whose status is not filed, scheduled, or controlled is found in the departure time array, an error message containing information about the offending flight is sent to the main *fdb_manager* process transcript pad. The incorrect entry is ignored, and array processing continues.

An error which occurs while enqueueing a transaction to the TDB will result in a fatal execution error. The *fdb_manager* process and all child processes will terminate, and a program traceback will be written to the transcript pad.

25.14.4 The Process Flight Messages Module

Purpose

Process Flight Messages receives parsed NAS, OMP, and flight schedule messages from the *Parser* and parsed EDCT messages from the *EDCT Server*. *Process Flight Messages* updates the *Flight Database* with data from these messages. It also predicts flight times and generates output flight messages for the *Traffic Demands Database Processor* and the *Flight Table Manager.* While performing these tasks, *Process Flight Messages* maintains the time-out arrays used by *Delete Old Flights* and *Remove Late Departures From TDB* and the hash table used to perform flight matching.

Design Issue: the Hash Table

In order to facilitate matching of the flight IDs contained in messages to those in the *Flight Database*, *Process Flight Messages* utilizes the hashing function and hash table depicted in Figure 25-21. This hashing function converts the flight ID for each message into an index to a hash bucket in the hash table. The hash bucket is a linked list of hash table records (**hashtab_rec_t**; see 25-7), each of which contains the address of the flight's record in the FDB, the flight ID, and the flight's status (**active, proposed**, etc.).

When *Process Flight Messages* receives a message, it calculates the index of the hash bucket for the flight ID in the message. *Process Flight Messages* then traverses the bucket's list of hash table records for FDB entries with the same flight ID. Each hash table record with the matching flight ID is stored in an output list, which is sorted by flight status before being sent to the appropriate message processing module.

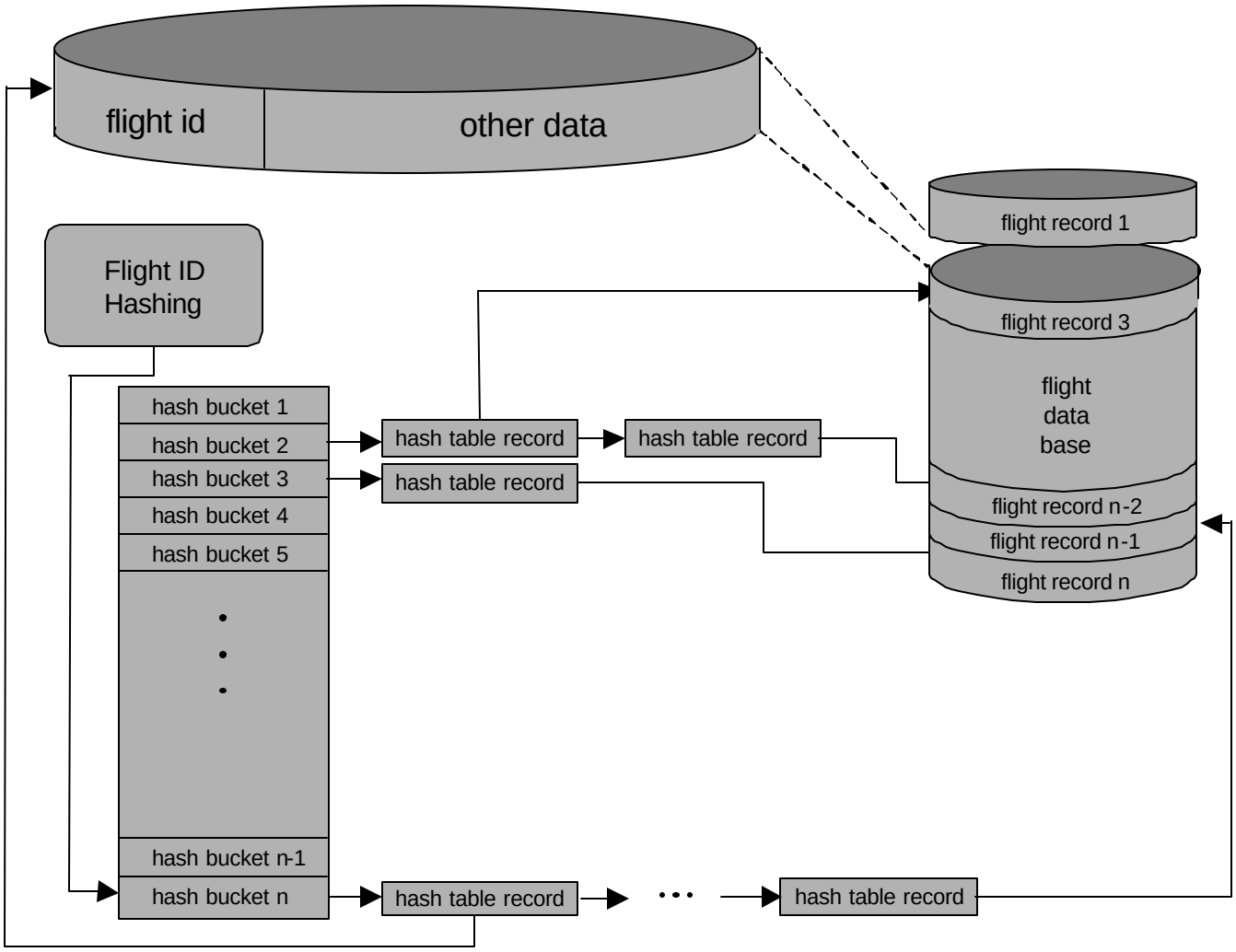

Figure 25-21. A Hashing Function and Hash Table Relates Flight ID to Address

Table 25–7. hashtab_rec_t Data Structure

hashrecord_t			
Library Name: fdb_openlib		**Element Name:** hash_structure.h	
Purpose: to store information in hash table linked list			
Data Item	**Definition**	**Unit/Format**	**Var. Type**
fdboffset	Offset of flight record into fdb map file.	Number of bytes from beg. of map file (0 to MAXFDBOFFSET)	INT32
nxtoffset	Pointer to next hash table record in linked list.	-	INT32
flight_id	Flight identification.	1 3 chars followed by numbers	string7
fstatus	Status of the flight.	Enumerated type: filed, active, etc.	status _of_flight

Upon receiving a flight message for which there does not exist a current FDB entry, *Process Flight Messages* adds the flight message information to a new entry in the FDB. At that time, *Process Flight Messages* hashes the flight's ID into an index and inserts a new hash table record for this flight into the linked list corresponding to the computed index.

Input

(1) Parsed NAS messages –– These contain the flight state and route data that have been drawn from NAS messages by the *Parser*. The data available in these messages can be found in Section 6. *Process Flight Messages* processes information from the parsed results of the following types of NAS messages: Departure (DZ), Arrival (AZ), Cancellation (RZ), Flight Plans (FZ), Boundary Crossing (UZ), Position Update (TZ), and Amendment (AF). Parsed DZ, AZ, and RZ messages contain flight state data. The parsed FZ and UZ messages usually include the event list representations of flight routes. Parsed TZ messages contain the last reported position, altitude, and speed of flight. Parsed AF messages might contain some combination of flight state and route data. Each message notes the time the ETMS received it.

(2) Parsed flight schedule messages –– These contain the flight state data and route data that have been extracted by the *Parser* from the *Schedule Database*-generated, flight schedule messages. The data available in these messages can be found in Section 6. There are two types of flight schedule messages for which the *Parser* sends data to *Process Flight Messages*: the FS (flight schedule) and the RS (cancel scheduled flight) messages. The parsed FS messages usually include the event list representations of flight routes.

(3) Parsed EDCT messages –– These contain the flight state data that have been extracted by the *EDCT Server* from the data files which comprise an Air Traffic Control System Command-generated ground delay program. Most importantly,

each parsed EDCT message contains the identification of the flight to be ground delayed, its departure airport, and its new controlled departure time. Section 6 contains more information about the EDCT messages.

(4) Parsed OMP messages –– These contain position updates for oceanic flights outside of CONUS radar coverage from the OMP. The data available in these messages can be found in Section 6. *Process Flight Messages* processes information from the parsed results of these Oceanic Position Update (TO) messages.

(5) Aircraft dynamics data –– The aircraft dynamics data includes information about the flight characteristics (speed, altitude, and rate of descent) of various types of aircraft. Section 23.1 contains more information about these aircraft dynamics data.

(6) Element names –– *Process Flight Messages* obtains element names from the element name file, which maps an element's type and index to its name. A more detailed description of the element name file can be found in Section 19.

(7) Wind data –– *Process Flight Messages* gets wind speed and direction from the *Grid Winds Database*, which is created by the *gridwinds_read* module (Section 25.14.1.2). The *Grid Winds Database* maps a geographic position and altitude into the wind speed and direction at that position.

(8) Old flight data –– *Process Flight Messages* receives old flight data from the flight database (Section 25.14). These data consist of the latest information about a flight before the current message has been processed. These data include the latest event list (including times), flight status, last actual event, and other flight-specific information.

Output

Process Flight Messages generates the following output:

(1) New flight data –– As *Process Flight Messages* successfully processes each input message, it writes the resulting new flight data to the *Flight Database*. This new data includes a new event list (with new times), new flight status, and other flight-specific information.

(2) Departure time updates –– These post-message processing updates to the departure time-out arrays include the flight's latest departure time, status, and address in the flight database.

(3) Arrival time updates –– These post-message processing updates to the arrival time-out arrays include the flight's latest arrival time, status, and address in the flight database.

(4) TDB transactions –– Event list and time information, used by the *Traffic Demands Database Processor* to track demands at NAS elements.

(5) Flight update messages –– Flight update information sent via the *FDB Distributor* to the *Flight Table Manager* for use by the *TSD* and route messages in version 4.2 format for the ASDI link.

(6) Feedback messages –– FA messages containing information gathered from NAS AF messages and information contained in the *Flight Database*.

Processing Overview

The *Process Flight Messages* module is made up of the following five modules: *Update FDB With Message Data, Compute Flight Times*, *Merge Event Lists*, *Generate Route Information*, and *Send TDB Transactions*. The data flow between these modules is shown in Figure 25-22.

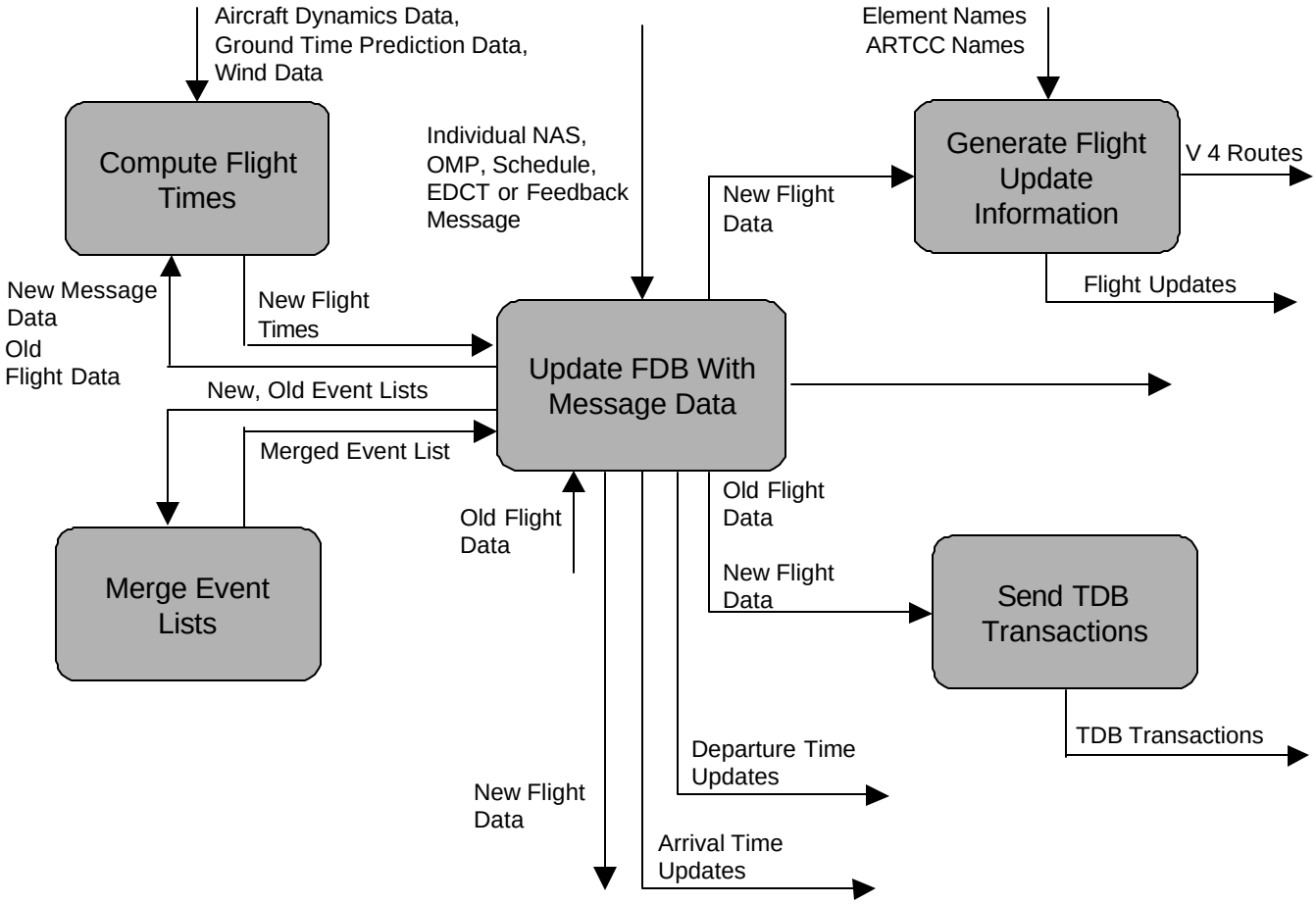

Figure 25-22. Data Flow of the Process Flight Messages Module

Depending on the type of message being processed, the *Process Flight Messages* module may not perform all functions described by the modules shown in Figure 25-22. However, this data flow diagram and the flow chart in Figure 25-23 describe the processing in general terms.

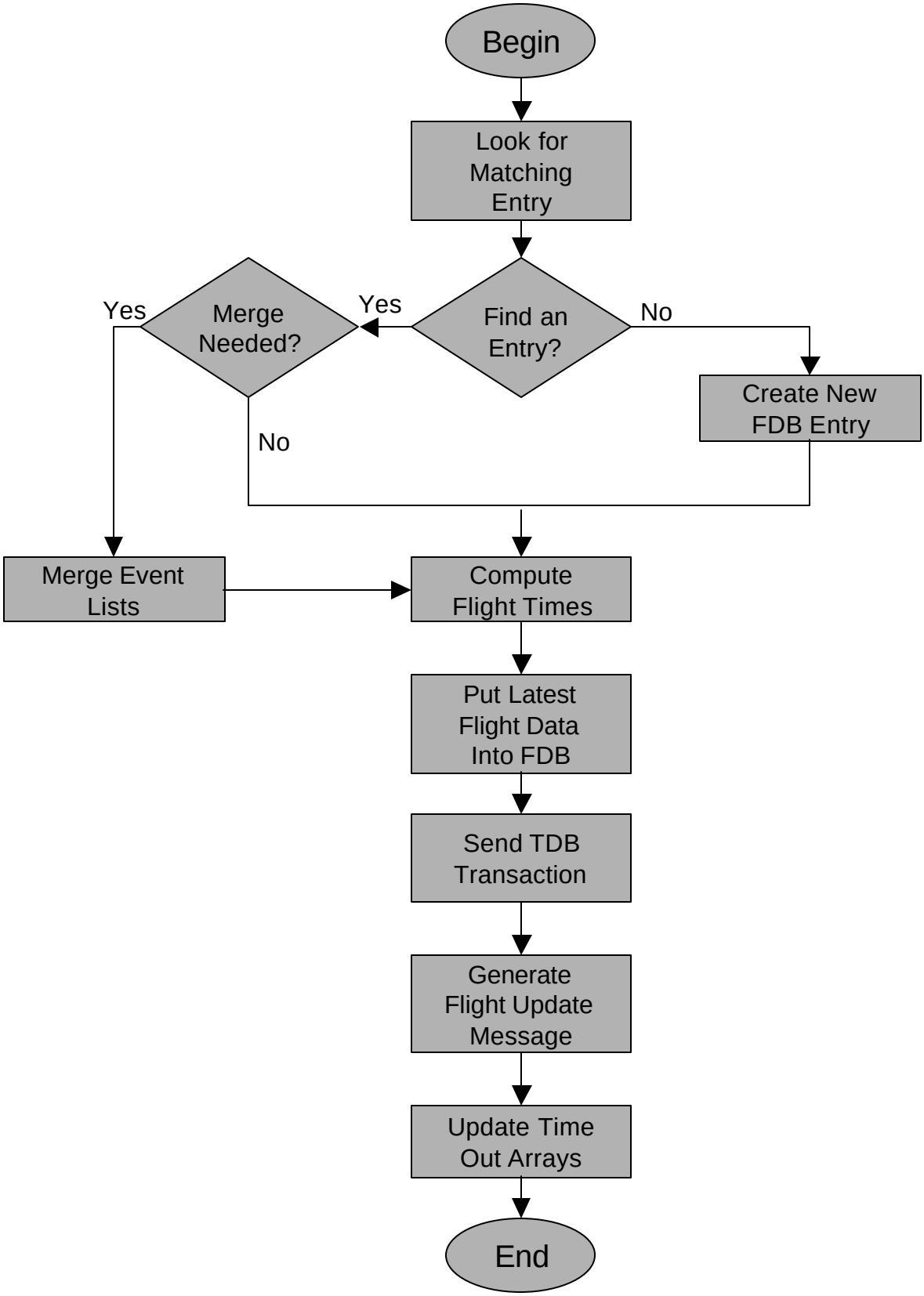

Figure 25-23. Flow Chart for Process Flight Messages

In general, the *Process Flight Messages* module processes NAS, OMP, EDCT, flight schedule, and FA messages in a manner as described in the flow chart . The *Process Flight Messages* module first dequeues a message which had been received by the *FDB Receiver* process. *Process Flight Messages* uses identifying information from the message (e.g., flight identification, departure and arrival airports) and a hash table to find a matching flight from the flight database. A description of the hash table is described in this section as a Design Issue.

If *Process Flight Messages* does not find a matching flight in the flight database, it creates a new slot in the flight database and fills the slot with information from the input message. If a matching flight is found, *Process Flight Messages* merges the data from the message into that flight's record. For messages with event lists, *Process Flight Messages* merges the new event list with the one existing in the flight database. Event-list merging is described in Section 25.14.4.2.

In either case (a match is found or not found), if the flight has an event list, *Process Flight Messages* computes the times of the flight at each of these events. This time computation is described in detail in Section 25.14.4.1. *Process Flight Messages* places the new or merged flight data, which includes the new times, into the flight database. The time-out arrays, described in Section 25.14.2, are updated using the new flight times.

For most message types, *Process Flight Messages* generates a TDB transaction which directs the *Traffic Database Processor* to add, replace, or delete this flight in its traffic demand counts database. The generation and transmission of these TDB transactions is described in Section 25.14.4.3. *Process Flight Messages* generates flight update messages which are sent to the *FTM*.

The general description of *Process Flight Messages* accompanying Figure 25-23 highlights processing performed in the case of some (sometimes all) message types. Yet, *Process Flight Messages* actually performs different functions depending on the type of message being processed. These message-specific modules (*Do_FZ, Do_UZ, Do_DZ, Do_RZ, Do_AZ, Do_AF, Do_TZ, Do_TO, Do_FS, Do_RS, Do_EDCT)* are described in the sections following the descriptions of the shared design issues and processing.

25.14.4.1 The Compute Flight Times Module

The *Compute Flight Times* module supports the main task of the *FDP*: the prediction of times for each event in a flight's event list. *Compute Flight Times* is itself a loose group of modules and routines that are applied selectively by each message processing module. The one exception is computation of flight times for position update (TZ) messages. Due to its complex nature, the TZ message processing module (*Do_TZ)* utilizes its own internal flight times computation routines, described in Section 25.14.4.5.6.

Under most circumstances, most message processing modules compute future flight times along the event list through the *ModelFlight* module (see Section 25.14.4.1.1). Two of *ModelFlight*'s supporting routines, *groundspeed_with_winds* (Section 25.14.4.1.2) and *get_time_value* (Section 25.14.4.1.3) are used respectively to supply winds-modified air speeds and flight-profile dependent event times.

One exception to the use of *Modelflight* for event times computation is found for all DZ (departure) messages, all EDCT (departure control time) messages, and proposed AF (flight plan amendment) messages which only request a change to the flight's departure time. These message types use the routine *add_event_list_delta_time* to more efficiently update event times.

Add_event_list_delta_time takes as input an event list, a delta time value, and a time type for the first event in the event list. It supplies as output an event list in which all times have been modified by the same value. Processing begins by placing the time type into the first event. The routine then traverses the entire event list, adding the delta value for each event in the event list.

Another such exception to the use of *Modelflight* is made for event lists where the message's coordination fix event is after the last actual event in the event list. UZs (boundary crossings), AF/FAs (flight plan amendments), active FZs (flight plans) and AZs can all fall into this category. For this instance, another event time computation algorithm is used to locate the coordination fix event in the event list and interpolate times for those events which are in the past. This module, called *interpolate_previous_flight_times*, is described in Section 25.14.4.1.4.

25.14.4.1.1 The ModelFlight Module

ModelFlight receives four types of input:

(1) Event list –– This list is a representation of a flight. Each event contains data describing the intersection of the flight with a NAS element.

(2) Aircraft dynamics data –– These data include the aircraft type, the ascent and descent profile, and other information such as altitude and speed of the flight.

(3) Winds data –– Contains wind data at fifteen different altitude levels by latitude and longitude over the contiguous United States.

(4) Flight start time –– This time is used as the time of the last non-modeled event in the list. Each subsequent event time is modeled with respect to this time. Usually, this time is a flight's departure time and is assigned to the first event in the list.

ModelFlight's output consists of a modeled event list, which is identical to the input list, except for the addition of event times.

Model Flight begins its processing by assigning a start time to the appropriate event in the list. It then predicts the times of all subsequent events in the following manner: for each event, *ModelFlight* extracts the previously computed distance between this and the previous event. For scheduled and proposed flights, if an event occurs during the en route phase of the flight, *Modelflight* uses the *groundspeed_with_winds* (see Section 25.14.4.1.2) routine to predict a winds modified ground speed for the event. *ModelFlight* then passes the computed distance value, along with the aircraft dynamics data and new ground speed, to the function *get_time_value*. The function *get_time_value* (see Section 25.14.4.1.3) uses the supplied data to determine the time of flight between this and the previous event. *ModelFlight* adds this relative time to the absolute time extracted from the previous event. The resulting absolute time is then included in the data structure that makes up the current event. This process is repeated for each event.

25.14.4.1.2 The groundspeed_with_winds Routine

Groundspeed_with_winds takes as input the flight's location for an event (latitude, longitude and altitude), its heading, its filed airspeed, and the time for the flight's previous event.

It returns as output a winds modified ground speed value based on the filed airspeed and the current wind value at the specified location and time.

In order to access the grid winds database, *Groundspeed_with_winds* converts the latitude, longitude, altitude, and time input parameters into grid winds database (GWDB) indices. The indices are generated in the following manner:

(1) Groundspeed_with_winds applies the formulas described in Section 25.14.1.2 to convert the longitude and latitude coordinates into Cartesian **x** and **y** values. It then performs binary searches on the GWDB **x** and **y** arrays to locate the array indices corresponding to the **x** and **y** values.. These array indices are the GWDB indices for the lat/lon.

(2) The routine searches an array of altitudes to find the index corresponding to the input altitude. Altitudes greater than the maximum altitude (flight level 450) are assigned the maximum altitude index.

(3) Groundspeed_with_winds compares the input time with those contained in the GWDB time control array. It locates the control record nearest in time to the input time and extracts the GWDB data set index from the control record.

If all the indices are within range, *groundspeed_with_winds* extracts the wind direction and wind speed from the grid winds database. *Groundspeed_with_winds* then applies a vectorial approach; it first calculates a modified heading for the flight based on the wind and the desired direction of travel. It then projects the airspeed along this modified heading to obtain a prediction of the flight's actual speed relative to the ground.

If, due to an error in the winds data or the flight's airspeed, the ground speed predicted by *groundspeed_with_winds* is less than 10% of the filed airspeed, the routine returns an error and discards the modeled ground speed value.

25.14.4.1.3 The get_time_value Routine

Get_time_value computes the flight time by first considering the vertical flight orientation (i.e., *ascending*, *flying leve*l, or *descending*) of the previous and the current flight event. The time leg of each orientation stage is computed with its own data structures. Since there are three orientations and two flight events, there are six possible combinations, as shown in 25-8.

Table 25-8. Flight Orientation Combinations

Previous Event	Current Event
ascending	ascending
ascending	flying level
ascending	descending
flying level	flying level
flying level	descending
descending	descending

The *get_time_value* routine uses the flight phase, altitude, speed, and distance along the flight path at two sequential events to model the time, in minutes, between the two events. *ModelFlight* calls this procedure (see Section 25.14.4.1.1) at each flight event along the flight path. These two routines communicate data via the **flight** record, the structure of which is described in Table 25-9.

Table 25-9. Flight Record Data Structure

Flight Record							
Library Name: Profile_openlib			**Element Name:** Profile.h				
Data Item	**Definition**	**Unit**	**Legal Range**	**Var. Type**	**I/O by function+**		
					F_1	**F_2**	**F_3**
flt_id	flight identifier (e.g., AAL123)			array[1…10] of char	|	--	--
filed_fz_onground	indicates disposition of the filed Field 10 Field 10 filed on grnd=T Filed in air=F		T or F	Boolean	|	--	--
civ	indicates if the aircraft is civilian=T or military=F		T or F	Boolean	O	|	--
runaway	indicates that flight errors or inconsistencies are severe & fatal=T No major problem=F		T or F	Boolean	O	--	--
dsg_actual	actual designator taken from the FZ			array[1…4] of char	|	--	--
aircraft_index	index indicating a record in the Aircraft_Descriptor Map which describes the given flight		-1 to max_plane_type	short	O	--	--
dsg_index	index indicating the particular template aircraft assigned to this flight		1 to tot_templates	short	O	|	|
ascent_index	index indicating the particular ascent profile for this flight		1 to max_ascent_profiles	short	O	|	|
descent_index	index indicating the particular ascent profile for this flight		1 to max_descent_profiles	short	O	|	|
dist_total	total distance for this flight	nautical miles		INT32	|	|	|
origin_lat	latitude of the originating airport	radians		float	|	|	|
origin_lon	longitude of the originating airport	radians		float	|	|	|
dest_lat	latitude of the destination airport	radians		float	|	|	|
dest_lon	longitude of the destination airport	radians		float	|	|	|
dist_cruz	distance from the takeoff roll to the point at which the aircraft achieves cruising altitude	nautical miles		INT32	O	|	|
spd_cruz	cruising speed for this flight	(nautical miles/min) x 100		INT32	|/O	|	|
alt_cruz	cruising altitude for this flight	feet/100		INT32	|/O	|	|

+ **F_1** indicates *Assign_A_Profile*, **F_2** indicates *Get_Altitude_Values*, **F_3** indicates *Get_Time_Value*

Table 25-9. Flight Record Data Structure (continued)

Flight Record (continued)							
Library Name: Profile_openlib			**Element Name:** Profile.h				
Data Item	**Definition**	**Unit**	**Legal Range**	**Var. Type**	**I/O by function+**		
					F_1	**F_2**	**F_3**
dist_descent	distance from the begin descent point to the point where the aircraft touches down	nautical miles	0 to 160	INT32	O	|	|
previous.time	accumulated time from takeoff roll to the previous location of this flight	minutes		float	--	--	|
previous.lat	latitude at the previous location of this flight	radians		float	O	|	|
previous.lon	longitude at the previous location of this flight	radians		float	O	|	|
previous.dist	previous distance along the flight path	nautical miles		INT32	O	|	|
previous.phase	flight phase at the previous location for this flight			flight phase see Sec.15	O	|	|
previous.alt	altitude at the previous location for this flight	feet/100	0 to 600	INT32	O	|	|
previous.speed	speed at the previous location for this flight	(nautical miles/min) x100		INT32	O	|	|
now.time	flying time from the previous location to the current location	minutes		float	--	--	O
now.lat	latitude at the current location of this flight	radians		float	|	|	|
now.lon	longitude at the current location of this flight	radians		float	|	|	|
previous.dist	current distance along the flight path	nautical miles		INT32	|	|	|
now.phase	flight phase at the current location of this flight			flight phase see Sec. 15	O	O	|
now.alt	altitude at the current location for this flight	feet/100	0 to 600	INT32	O	O	|
previous.speed	speed at the previous location for this flight	(nautical miles/min) x100		INT32	O	|	|
no_descent	indicates whether or not the descent is modeled. ne_descent_modeled=T descent is modeled=F		T or F	Boolean	|	|	|
no_descent	indicates whether or not the flight must land on this call. Must land now=T, not land now=F		T or F	Boolean	|	|	|

+ **F_1** indicates *Assign_A_Profile*, **F_2** indicates *Get_Altitude_Values*, **F_3** indicates *Get_Time_Value*

Get_time_value computes the time legs appropriate to the flight event combination and totals them to produce a flight time interval. The time leg computations are described in the following paragraphs.

During ascent, the flying time is computed using the distances, the ascent index, and the ascent maps. The value of the **previous.distance** is used to find an accumulated time value from the **ascent_by_dist** map. Similarly, if the current distance (i.e., **now.distance**) is within the ascent stage, that distance is used to find a second time value. The time interval between the previous point and the current point is computed as the difference between the two accumulated time values. If the **current.distance** is beyond the leveling out point (i.e., **dist_cruz**), the value of **dist_cruz** is used to find the second accumulated time value. The difference between **dist_cruz** and the **current.distance** is divided by the cruise speed to compute the level time leg. The total time interval represents the sum of the ascent and level time legs.

When a flight is flying level, the flying time computation is not quite as simple. If the previous and current events occur in the same phase, the time is calculated by dividing the common speed into the difference between the **previous** and **now** distances. However, if the previous event occurs in the **level_out** phase (i.e., within the original Terminal Control Area [TCA]) and the current event occurs en route, the procedure considers the previous and current aircraft positions and speeds, as well as the TCA boundary, to estimate the time of that leg.

For jet aircraft the times for descent segments are computed similarly to the ascent segments. The previous and current distances are used to find time values from the **descent_by_dist map**. The difference between the two time values produces the descent time leg.

Since the speeds of propeller-driven aircraft vary considerably, these aircraft are modeled somewhat differently. There is only one descent profile for all propeller-driven aircraft; this profile constitutes the fastest of the propeller-driven planes. When the value of the cruise speed is below the values of the descent profile, the time for that portion of the descent is computed using the cruise speed and distance. When the cruise speed is above the profile speed (i.e., closer to the destination airport) the time values extracted directly from the map are employed to compute the descent time leg. The descent times are computed by a procedure named *get_descent_time*, which is called by *get_time_value*. See Figure 25-24 for an illustration of the logic used in *get_time_value* and Figure 25-25 for the logic used in *get_descent_time.*

25.14.4.1.4 The interpolate_previous_flight_times Module

This module is used when message processing requires the computation of flight times for events that are in the past (based on the value of the message's coordination fix event). There are two possible processing methods depending on the message type.

In the case of an AZ (arrival) message, the location of the coordination fix in the event list is known (the last event). Therefore, the previous flight times computation traverses the event list from the last event to that last actual or proposed event. *Interpolate_previous_flight_times* first examines the event list to make sure that a previous proposed or actual time exists. If the previous time is proposed, the *interpolate_previous_flight_times* applies a delta time to each event from the previous event to the current event. If the previous time is actual, a distance weighted time value is entered for each event from the previous event to the current event.

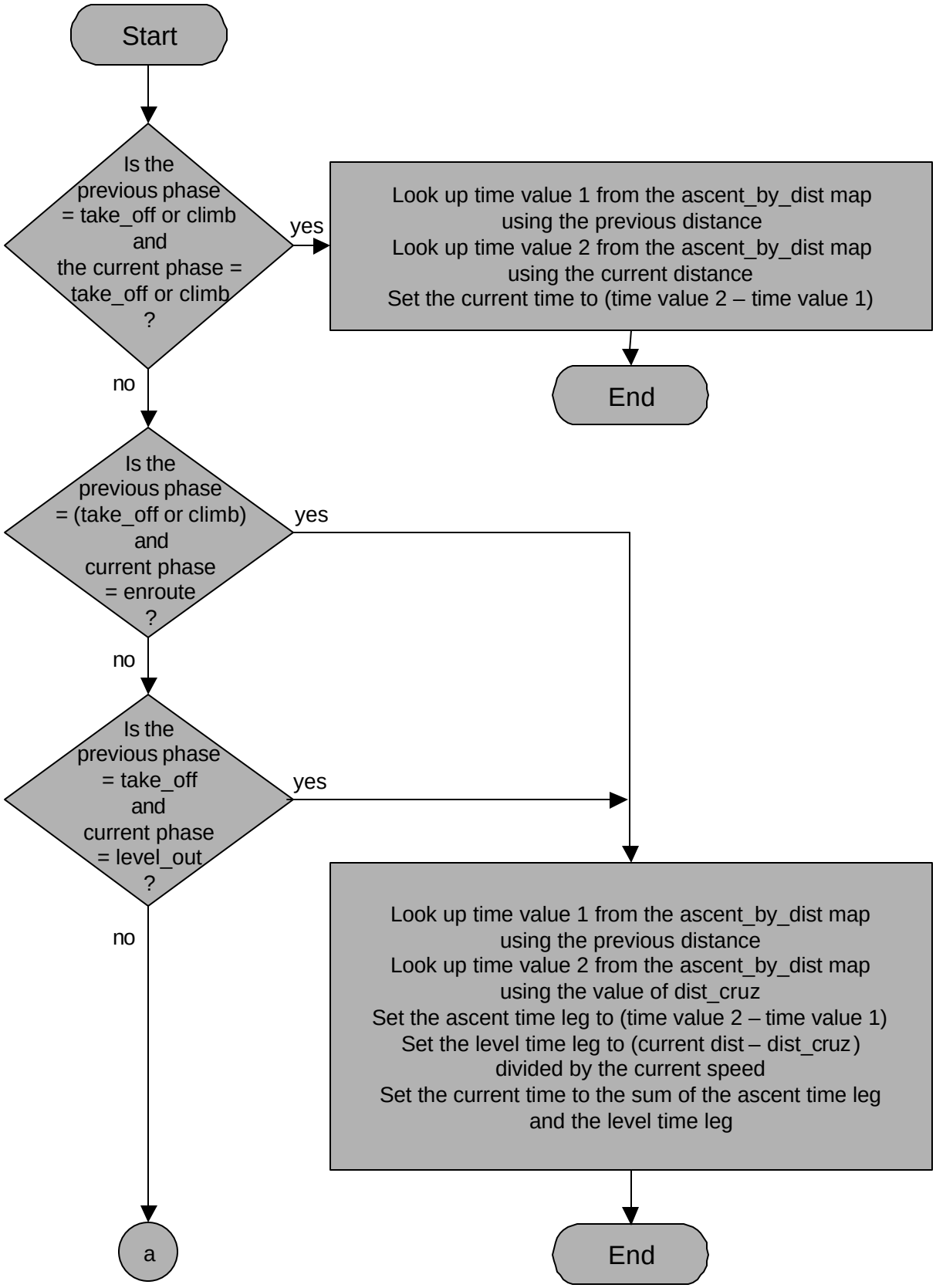

Figure 25-24. Sequential Logic for get_time_value Routine

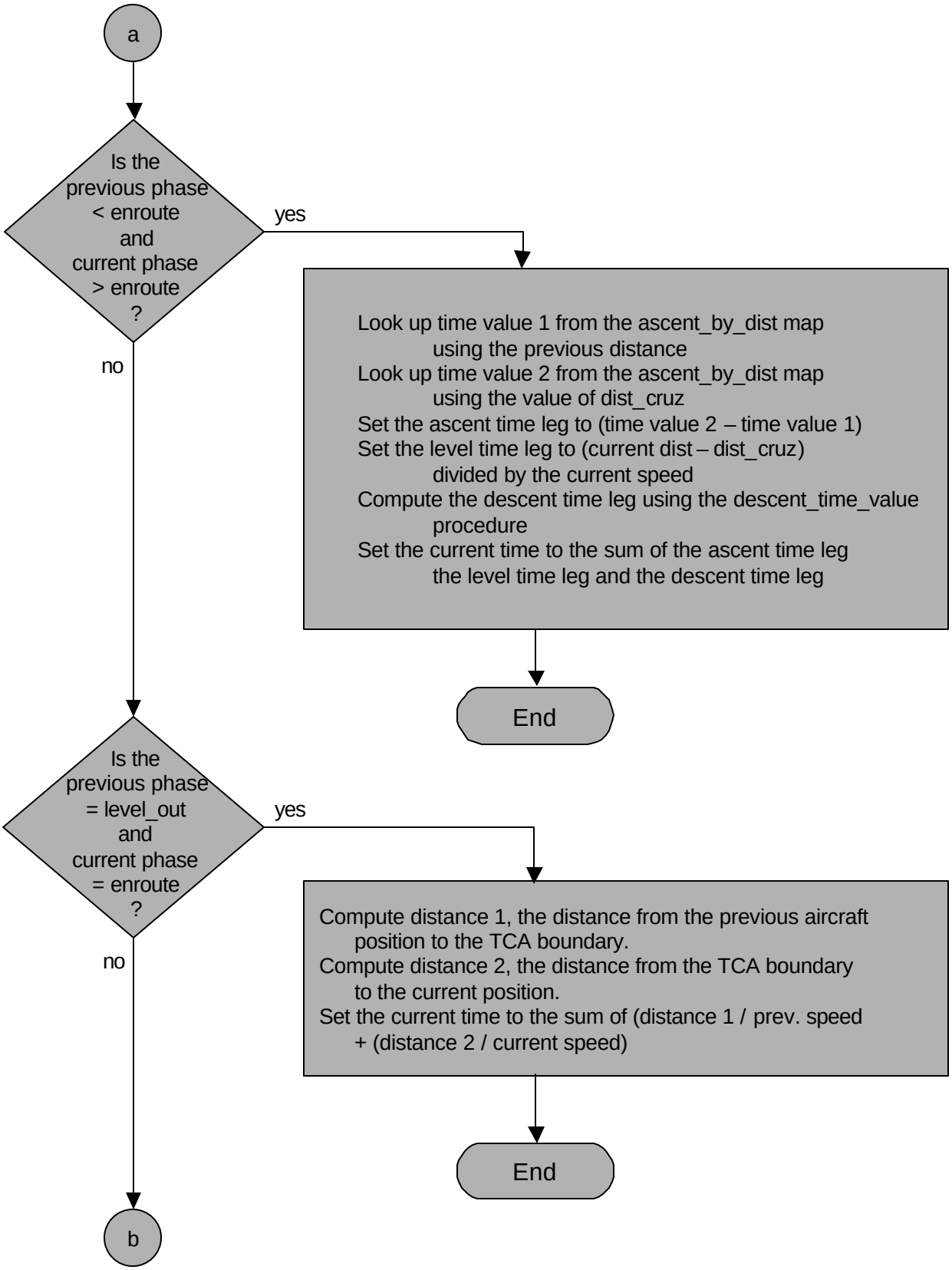

Figure 25-24. Sequential Logic for get_time_value Routine (continued)

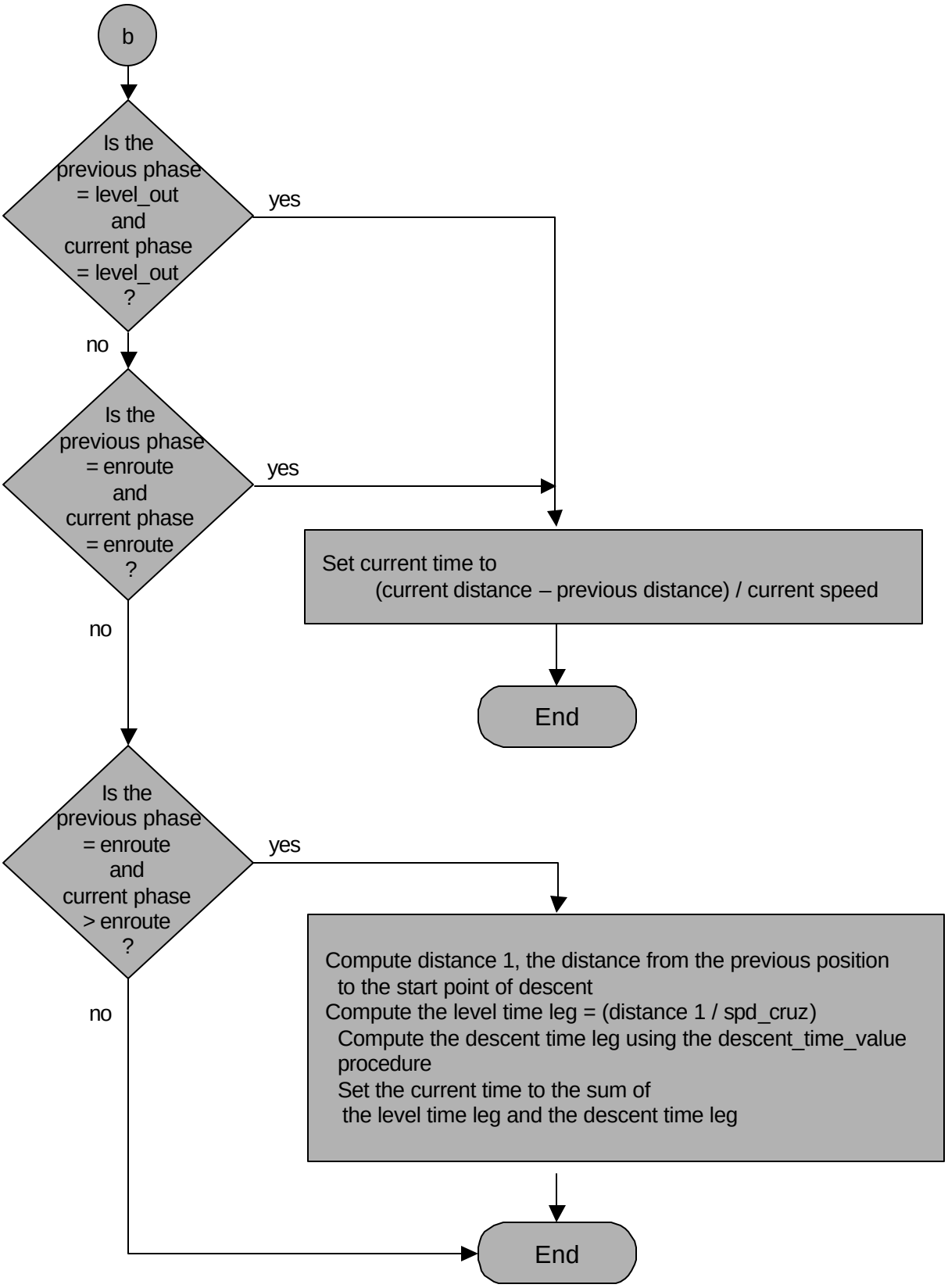

Figure 25-24. Sequential Logic for get_time_value Routine (continued)

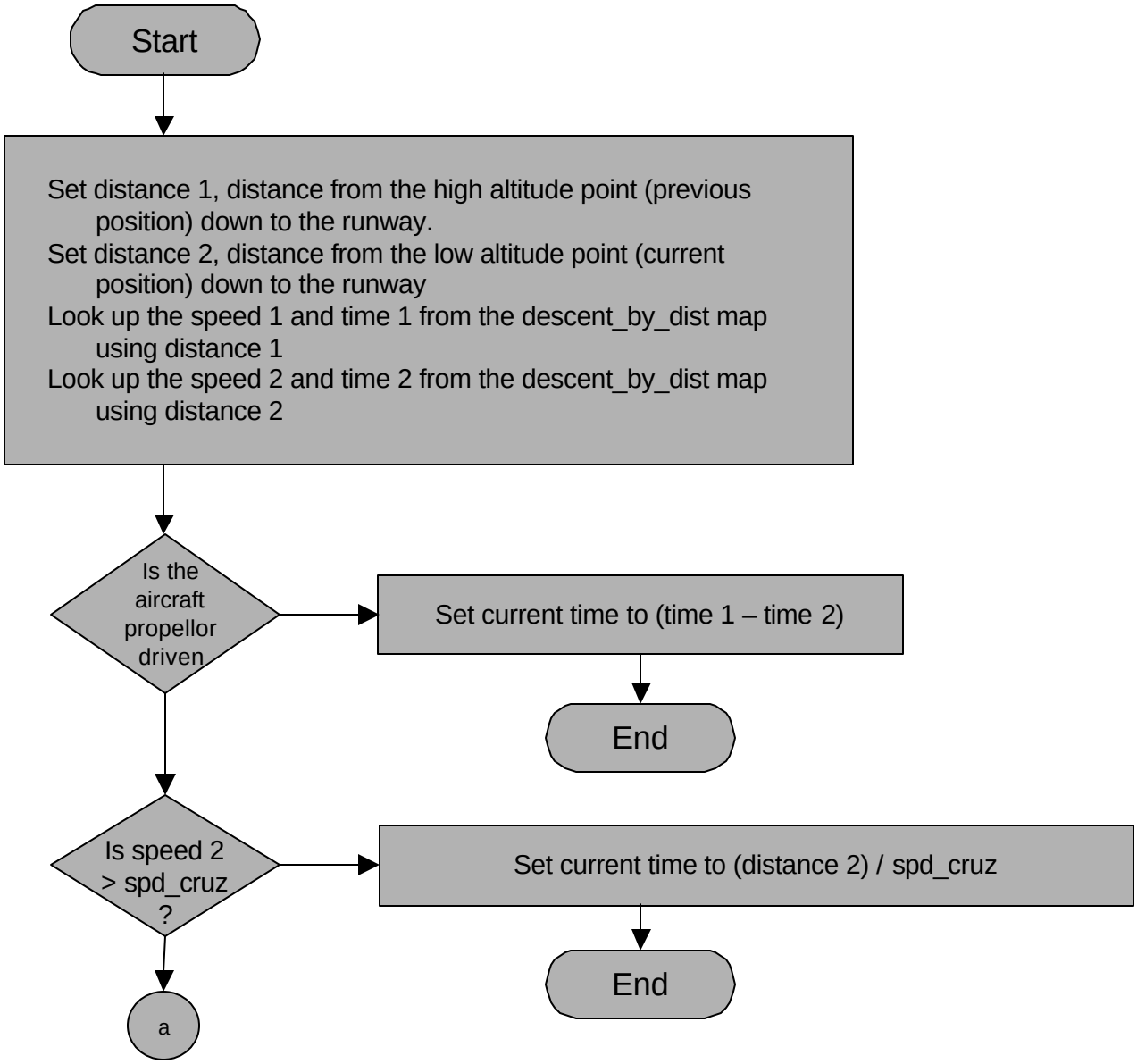

Figure 25-25. Sequential Logic for get_descent_time Routine

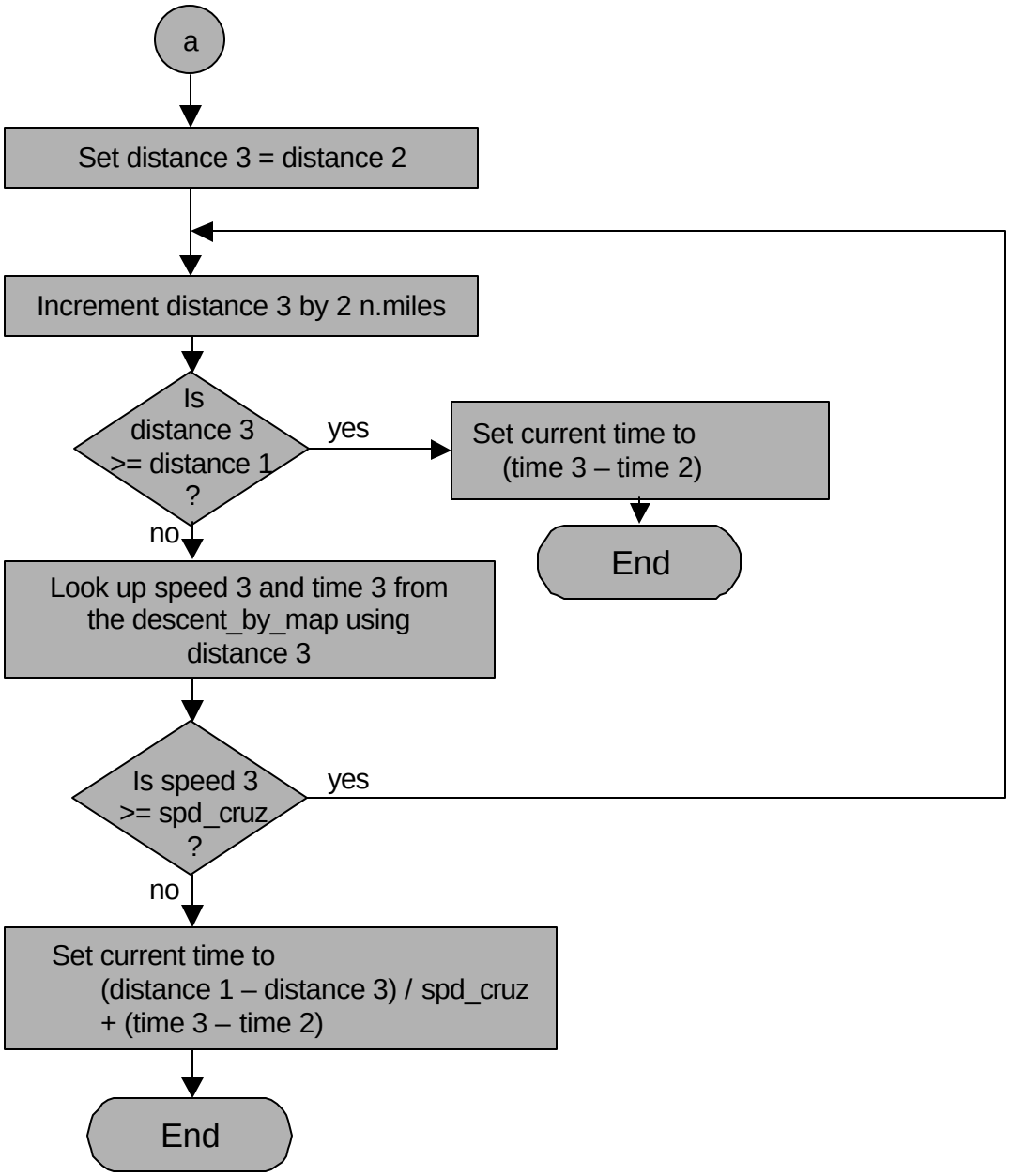

Figure 25-25. Sequential Logic for get_descent_time Routine (continued)

For all other message types, the coordination fix must be compared to the event list to locate the event that is closest. This comparison traverses the event list and determines which event is closest to the coordination fix event. Once the minimum distance event is located, *interpolate_previous_flight_times* checks to see if this minimum distance is within a tolerance. Depending on the result of this comparison, flight-times computation proceeds as follows.

If no event is found to be within the tolerance, *interpolate_previous_flight_times* assumes that the coordination event is before the beginning of the event list (truncated event list case). It then calculates the time for the first event in the event list based on the flight's average speed and the position defined by the coordination fix event. Since all events in the event list are in the future for this case, *Modelflight* is called to compute the flight times.

If an event is located within the tolerance, *interpolate_previous_flight_times* utilizes this event as the **last_actual_event**. It first uses the coordination fix event and the events before and after it in the event list to perform a distance and velocity weighted calculation of the time of the **last_actual_event**. *Interpolate_previous_flight_times* then uses *Modelflight* to compute the future flight times, constraining the time value at the **last_actual_event** to the previously calculated value. The resulting event list has the following qualities:

- Times which are interpolated previous to the **last_actual_event**.
- A time for the **last_actual_event** that is exactly equal to the estimated value.
- Future times computed by *Modelflight* which are consistent with the time in the **last_actual_event**.

25.14.4.2 The mergeeventlists Routine

The *mergeeventlists* routine takes two event lists and combines them such that all the points in the newer list are preserved. The old list is typically the full flight plan (FZ). The new list may be the result of a boundary crossing (UZ), which often begins in the midst of the old list. A new list may also be the result of an amendment (AF), and may be very different from the old list.

Mergeeventlists receives three types of data:

(1) Old event list.

(2) New event list.

(3) Flight status flag indicating whether flight is active or not.

Mergeeventlists produces three types of data as well:

(1) Merged event list.

(2) Pointer to the last actual event in the merged event list, i.e., the last event that we know has already occurred.

(3) Flag that indicates the strategy used to perform the merge. The codes are

  (a) 100: used strategy (1), special case, looping flights (see (1) below).

  (b) 200: used strategy (2), connect at a common point (see Figure 25-26).

  (c) 300: used strategy (3), connect at a close point (see Figure 25-27).

  (d) 400: used strategy (4), connect at a point based on reasonable bearings (see Figure 25-28).

  (e) 500: default, keep first event of old list and append new list.

The main task associated with combining two event lists is to determine the exact point at which to join the two lists. *Mergeeventlists* tries four successive strategies, and the first appropriate one is implemented. If none of the four methods can be implemented, the default merge is used. Afterwards, a flag is returned indicating which method was used. The strategies, corresponding to output flags, are presented as follows:

(1) Looping flights - A special case common to military flights where the old list consists of only an airport, and the new list starts with a fix and ends with the same airport. We simply concatenate the two lists.

(2) Connect at a common point - In many cases, there is a fix event common to both lists. This strategy is to look for such a fix and connect the two lists at that point.

First, procedure *matchfix* is invoked to find a common fix. This routine is passed the first point in the new event list and the entire old event list. It considers the fix to be present in the event list if it explicitly appears in the list or a fix event in the list is found which has the same or almost the same latitude and longitude. If a common fix is found, then the two lists are traced starting at that point to find the last actual event that they have in common. The two lists are connected at this point. In Figure 25-26, the last actual event could be **C**, **D**, or **E**. Assuming it is **D**, the merged list is then created by joining events in the old list up to, but not including, the last actual event (**A** to **C**) with events in the new list, starting with the last event (**D** to **J**).

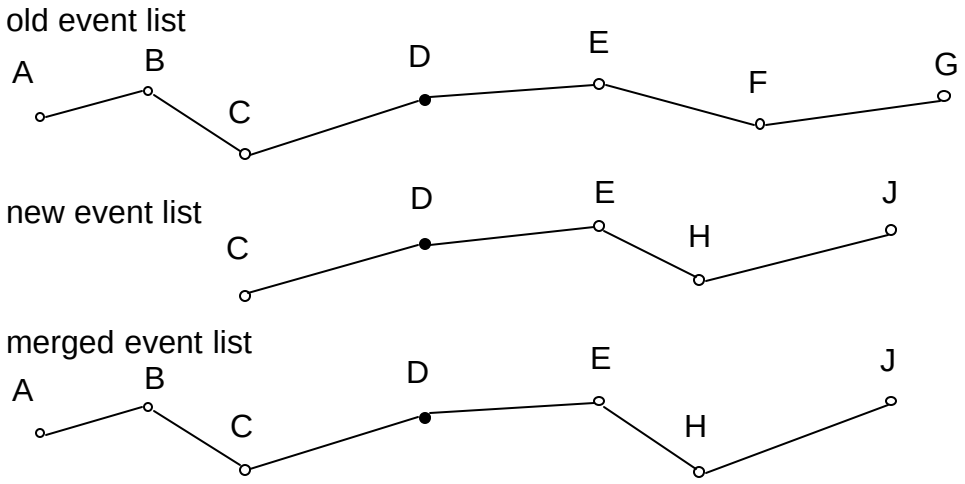

Figure 25-26. Merging Event Lists with a Common Point

(3) Connect at a close point –– If there is no common fix, then the routine tries to connect the beginning of the new event list to a point in the old list which is fairly close to it.

Function *flep* is used to determine which point on the old event list is closest to the beginning of the new event list. If the distance between the two points is more than twenty miles, then the next step is strategy (4). If not, then the routine connects the two lists at this point. In Figure 25-27, *flep* decides that event **C** is the closest point in the old list to event **H.** The merged list is created by joining events in the old list through the close point (**A** to **C**) with all events in the new list (**H** to **G**). Additionally, the **distance** and **heading** fields of event **H** are altered to reflect the distance and bearing of the new segment from **C** to **H**.

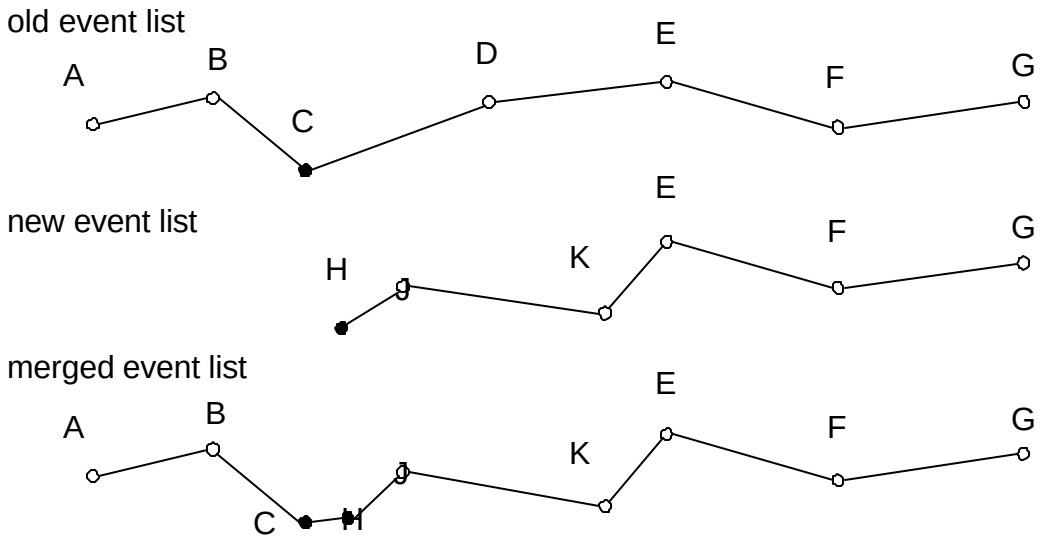

Figure 25-27. Merging Event Lists By Finding a "Close" Point

(4) Connect based on reasonable bearings –– If there is no common point, and no point on the old list is "close" to the beginning of the new list, then the routine tries to connect a point in the old list with the new list such that the resulting merged list is "smooth".

For example (see Figure 25-28), assume that event **C** is the last actual event. From **C,** the heading to event **J** differs greatly from the general heading to destination **G**. However, the heading from **A** to **J** is within 45_ of the general heading. When a point in the old list such as **A** is found, we create the merged event list by joining the old list through this point, with the new list. Again, the **distance** and **heading** fields of event **A** are altered to reflect the distance and bearing of the new segment from **A** to **J**.

NOTE: The list is traversed starting at the last actual event and moving backwards checking each waypoint for a valid merge point. If one is not found, the list is traversed starting after the last actual event and moving forward checking each waypoint for a valid merge point.

The *cleanup_merged_list* routine is invoked to make sure that after all the various manipulations, sector, route, and ARTCC entry and exit events still match up. Any that do not are deleted. Finally, the routine *cleanup_flightphases* cleans up out-of-sequence, phase information created by the merging process. *Cleanup_flightphases* also verifies that altitude and velocity information during the transition to the flight's en route phase is consistent.

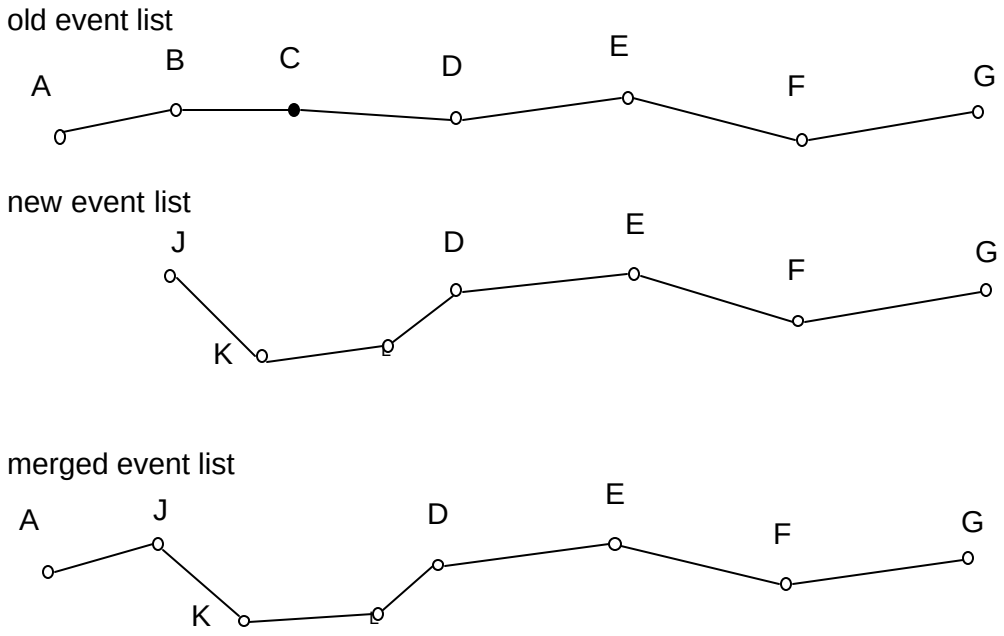

Figure 25-28. Merging Event Lists by Connecting the Last Actual Event

Note that if either event list is empty, results will be unpredictable. *Mergeeventlists* is not supposed to be invoked in this case.

25.14.4.3 The Send TDB Transactions Module

Depending on the type of input message, the new flight data in the message, and any existing flight data in the flight database, *Process Flight Messages* generates one of three types of TDB transactions. The three types of TDB transactions (**add_flight**, **replace_flight**, and **delete_flight**) direct the *Traffic Demands Database Processor* to add, replace, or delete the flight's effect on traffic demand counts. *Process Flight Messages* enqueues these transactions to the *TDB Relay*, which sends the transactions to the *Traffic Demands Database Processor*.

The *Send TDB Transactions* function does the actual generation and enqueuing of these transactions. When creating an **add_flight** transaction, *Send TDB Transactions* packs the new flight identification, flight status, and the newly modeled list of events into the transaction. For generating a **delete_flight** transaction, *Send TDB Transactions* packs the flight identification, old flight status, and the old event list into the transaction. In a **replace_flight** transaction, the unique flight identification is joined by (among other data) the old and new flight status, the old and new departure date, and the old and new event list.

In order to increase throughput to the TDB, *Send TDB Transactions strips out* any unnecessary data from these transactions. For all three types of transactions, *Send TDB Transactions* removes events from the event list for elements that are not monitored. In the case of a **replace_flight** transaction, *Send TDB Transactions* removes pairs of events (one from the old list, one from the new) if they are identical; they will cause no change in the traffic demand counts.

25.14.4.4 The Generate Flight Update Information Module

Each time a flight update occurs, the *Process Flight Messages* module prepares an update message and enqueues it to the *FDB Dist Relay*, which via *FDB Dist* sends the packet to the *Flight Table Managers* which have registered for services.

After NAS, OMP, schedule, and EDCT message data has been processed according to type, the update databases flag is checked to see if it was set to TRUE during the processing. If it was, the *Generate Flight Update Information* Module packs an update message. The type of data that gets packed depends on the type of message processed. A **Route Update** is generated when an AF, FA, FZ, FS, or UZ message is processed. A **Tz Update** is generated when a TZ message is processed. A **Time Update** is generated when an AZ, DZ, or EDCT message is processed. A **Cancel Update** is generated when an RZ, RS, SI_CANCEL_FLIGHT, or a CONTROL_CANCEL message is processed, and a **Position Update** is generated when a TO message is processed. After the data is packed, it is sent to *FDB Dist Relay* , which sends it to *FDB Dist*. *FDB Dist* then forwards the data to all connected *Flight Table Managers*. For messages with route information, a message is also sent to the *Route Relay* in version 4.2 format to be forwarded to the ASDI link.

25.14.4.5 The Update FDB With Message Data Module

The *Update FDB With Message Data* module directs the processing of each input message. The *Update FDB With Message Data* module consists of the message-specific processing performed by the following modules: *Do_FS, Do_FZ, Do_UZ, Do_DZ, Do_TZ, Do_RZ, Do_AF, Do_AZ, Do_EDCT, Do_TO,* and *Do_RS*. These modules are described in the following sections.

25.14.4.5.1 The Do_FS Module

The sequential logic diagram of Figure 25-29 highlights the fact that the *Do_FS* module can only add FS message information to the FDB; no replacement or deletion of flight information is performed by the *Do_FS* module. Thus, the primary concern of FS message matching is to detect and prevent the addition of redundant scheduled flight plans.

After the message flight ID is checked against the appropriate bucket in the hash table, *Do_FS* uses the flight status sorted list of flight addresses to attempt a match with **scheduled**, **filed, controlled,** or **active** database entries, in that order. If an exact match is found for any of these cases, *Do_FS* issues an error message and returns without processing the message. If no match is found, the scheduled flight plan is added to the database using the sequence of steps shown in Figure 25-30.

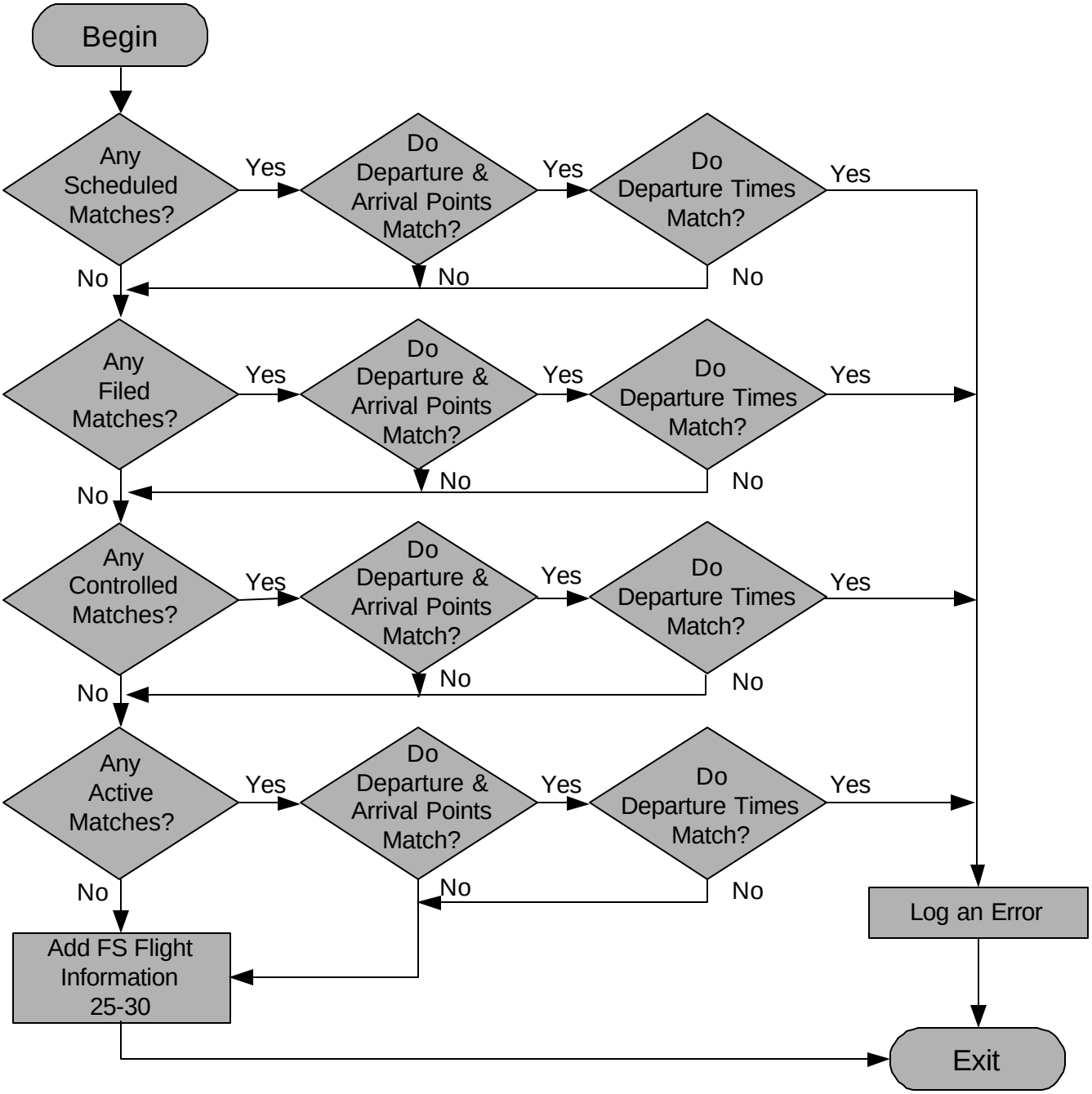

Figure 25-29. Sequential Logic for the DO_FS Module

In order to add a schedule flight plane to the database, *Do_FS* first allocates a new flight record and transfers the information from the message to the new flight record. Next, flight times are computed. The update databases flag is set to TRUE, and the TDB update type is set to add.

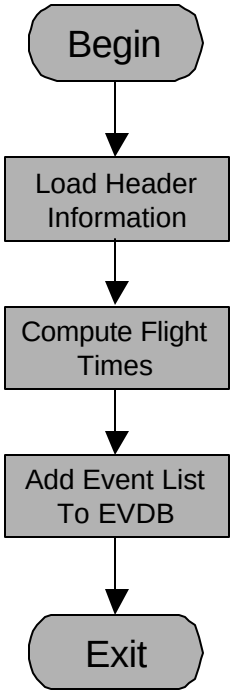

Figure 25-30. Sequential Logic for the Add FS Flight Information Routine

25.14.4.5.2 The Do_FZ Module

Detailed sequential logic diagrams for the message matching performed by *Do_FZ* can be found in Figure 25-31. Except where noted, the following discussion assumes that if a matching FDB entry is found, the flight information is updated as described later in this section and, if no match is found, the flight plan is added to the database.

When an FZ (flight plan) message is received by *Process NAS Messages*, an attempt is first made to match the flight ID in the message to flight ID's of records contained in the FDB. Flight ID matching is accomplished by means of the hashing function described in Section 25.14.4. If the flight ID match is successful, *Do_FZ* receives a list of all matching flights sorted by flight status. Depending on the type of message (active or proposed), *Do_FZ* searches this list in one of two orders of flight status based priority.

> **NOTE:** A proposed message is one with a computer identification code or one that originates outside CONUS. An active message is one without a computer identification code that originates in CONUS.

For a proposed message, *Do_FZ* first attempts a match with a **controlled** flight; failing this, it proceeds to check matches with **scheduled** flights, matches with computer IDs, **active** flights, **cancelled flights**, and finally **filed** flights. If a computer ID match is found to a flight which is already **active**, *Do_FZ* logs an error and discards the message. If the message matches a **filed** flight without matching that entry's computer ID, *Do_FZ* treats the entry as a multiple flight plan. When this occurs, *Do_FZ* checks all matching entries to see which one is currently in the TDB, deletes any existing TDB entry for the flight, and adds the new flight plan to the database. *Do_FZ* also preserves any controlled or scheduled times present in the old flight entry so that these times may be added to the new entry.

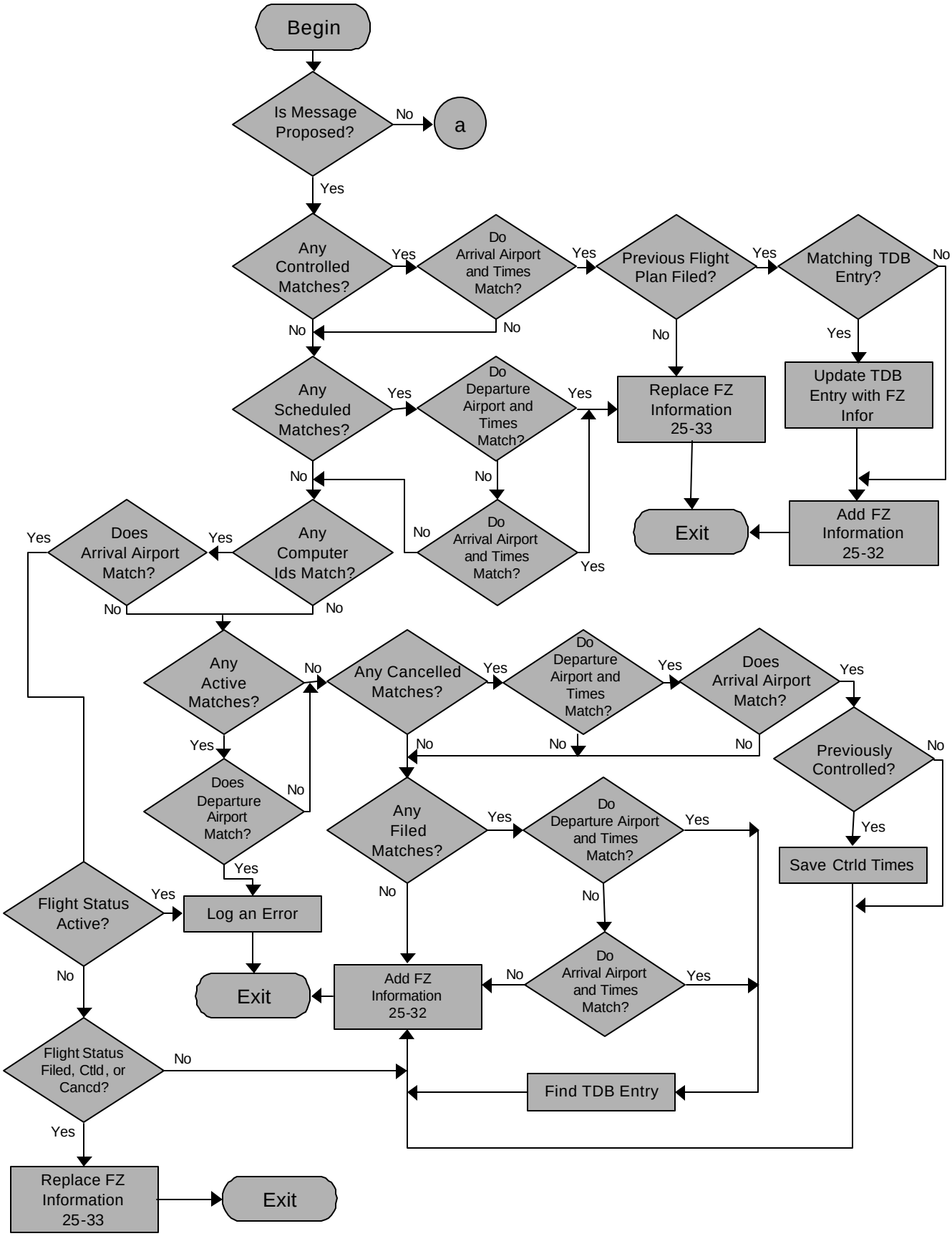

Figure 25-31. Sequential Logic for the Do_FZ Module

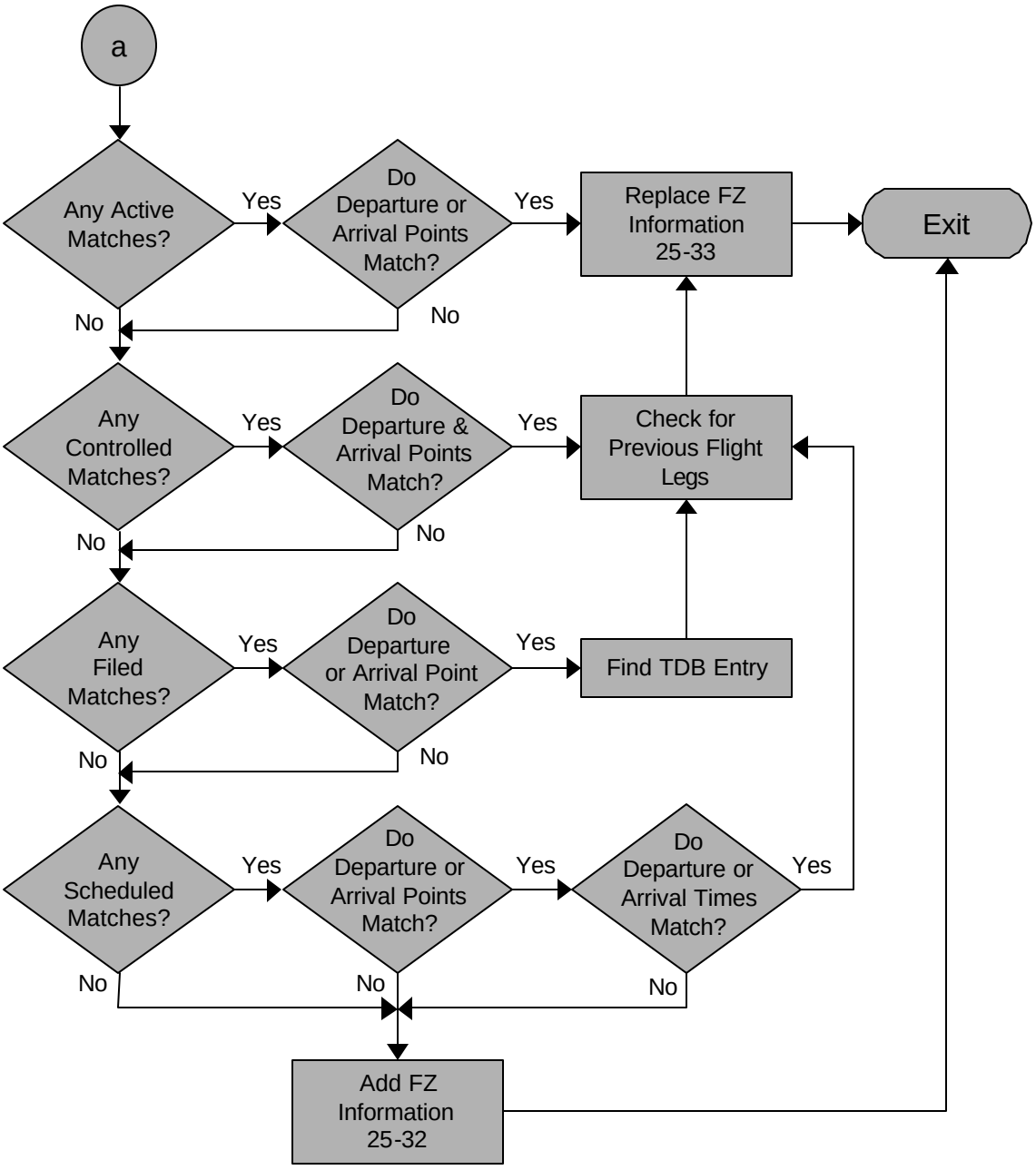

Figure 25-31. Sequential Logic for the Do_FZ Module (continued)

For an active message, *Do_FZ* checks for **active**, **controlled**, **filed**, and **scheduled** matches (in that order). A match to a **filed** flight with multiple flight plans requires that the update be performed on the entry which is currently in the TDB. In the case of a match to a **controlled, filed**, or **scheduled** flight, the list of active entries is first scanned by *check_previous_flightlegs* in order to locate and deactivate any **active** flights with the same flight ID as the message.

In order to add a new flight plan to the database (see Figure 25-32), *Do_FZ* first allocates a new flight record and fills in the header record information (proposed departure and arrival times, flight profile information, proposed altitude, and velocity, etc.). If the message is proposed, *Do_FZ* computes the flight times in the event list and adds the event list to the **evdb**. If the

message is active, *Do_FZ* examines the message's event list in order to locate the event that most closely corresponds to the coordination fix. After locating the coordination fix, *Do_FZ* interpolates previous event times and computes event times over the future portion of the event list. The update databases flag is set to TRUE, and the TDB update type is set to add.

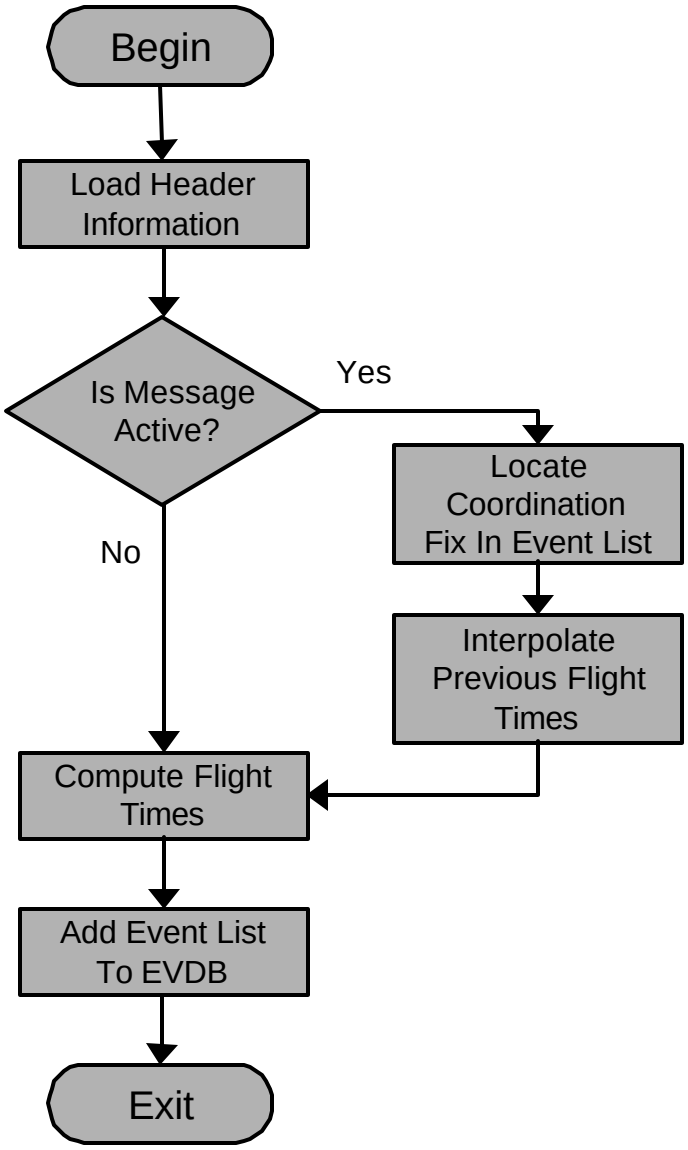

Figure 25-32. Sequential Logic for the Add FZ Information Routine

If a matching flight was found in the database, *Do_FZ* follows the logical flow shown in Figure 25-33 to update the database with the information contained in the message. An active message's event list is merged with the existing event list, and times are computed. If the merge fails, or if the message is proposed, *Do_FZ* replaces the existing event list with the one found in the message. The update databases flag is set to TRUE and the TDB update type is set to replace.

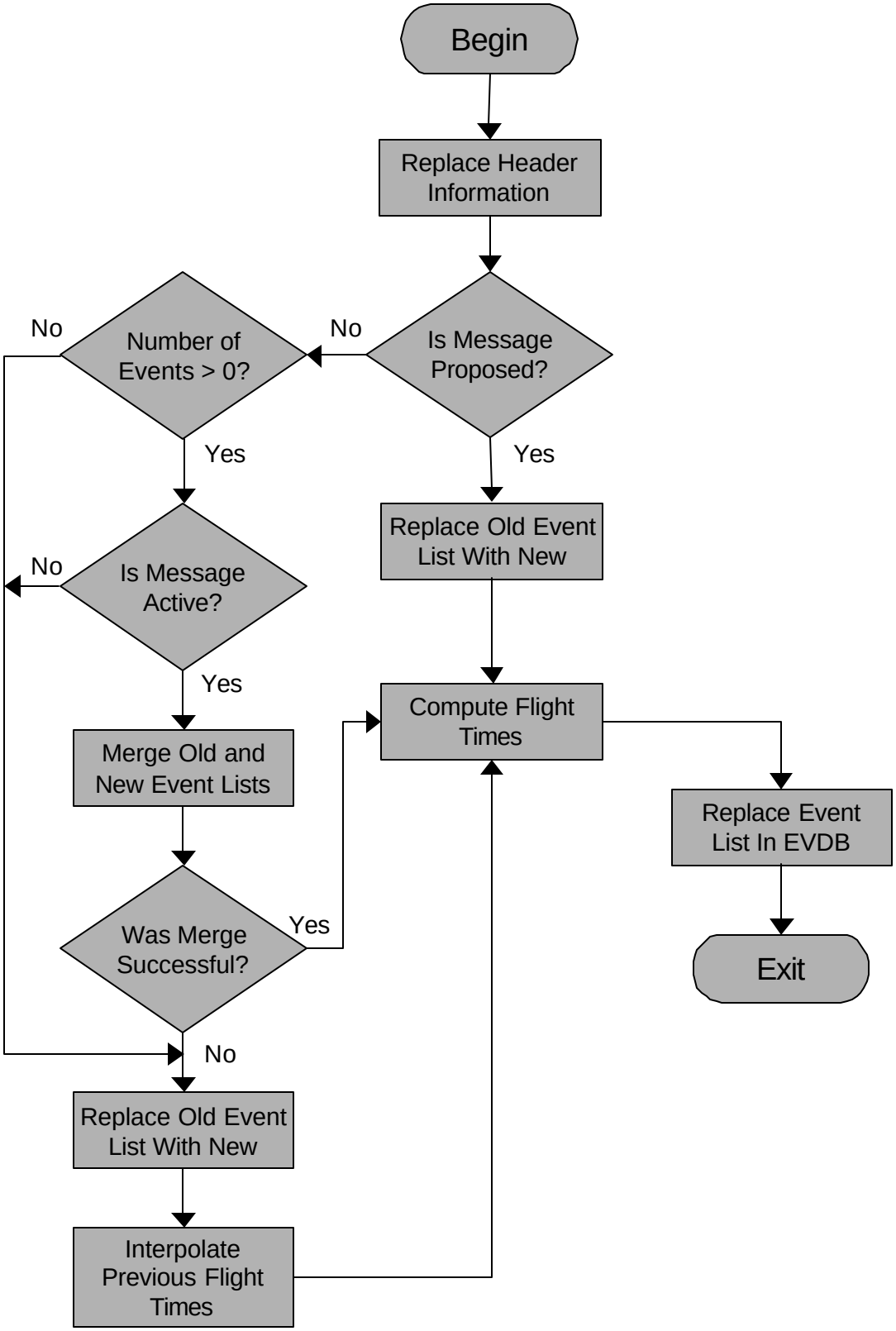

Figure 25-33. Sequential Logic for the Replace FZ Information Routine

25.14.4.5.3 The Do_EDCT Module

The *Do_EDCT* module processes EDCT (Estimated Departure Control Time) messages as shown in Figure 25-34. For each EDCT message, *Do_EDCT* receives a list of existing flight entries (sorted by flight status) with the same flight ID as the message. *Do_EDCT* searches this list for the entry which best matches the EDCT message in the following flight status order: **filed**, **scheduled, active**, **controlled**. For **filed**, **scheduled**, and **controlled** matches, *Do_EDCT* also checks for flights that have the same departure and arrival points as the EDCT message before updating the database entry with data from the message. If an **active** database entry with same departure and arrival points, the message is ignored. When no match is found through the process described above, *Do_EDCT* adds the EDCT information to the database.

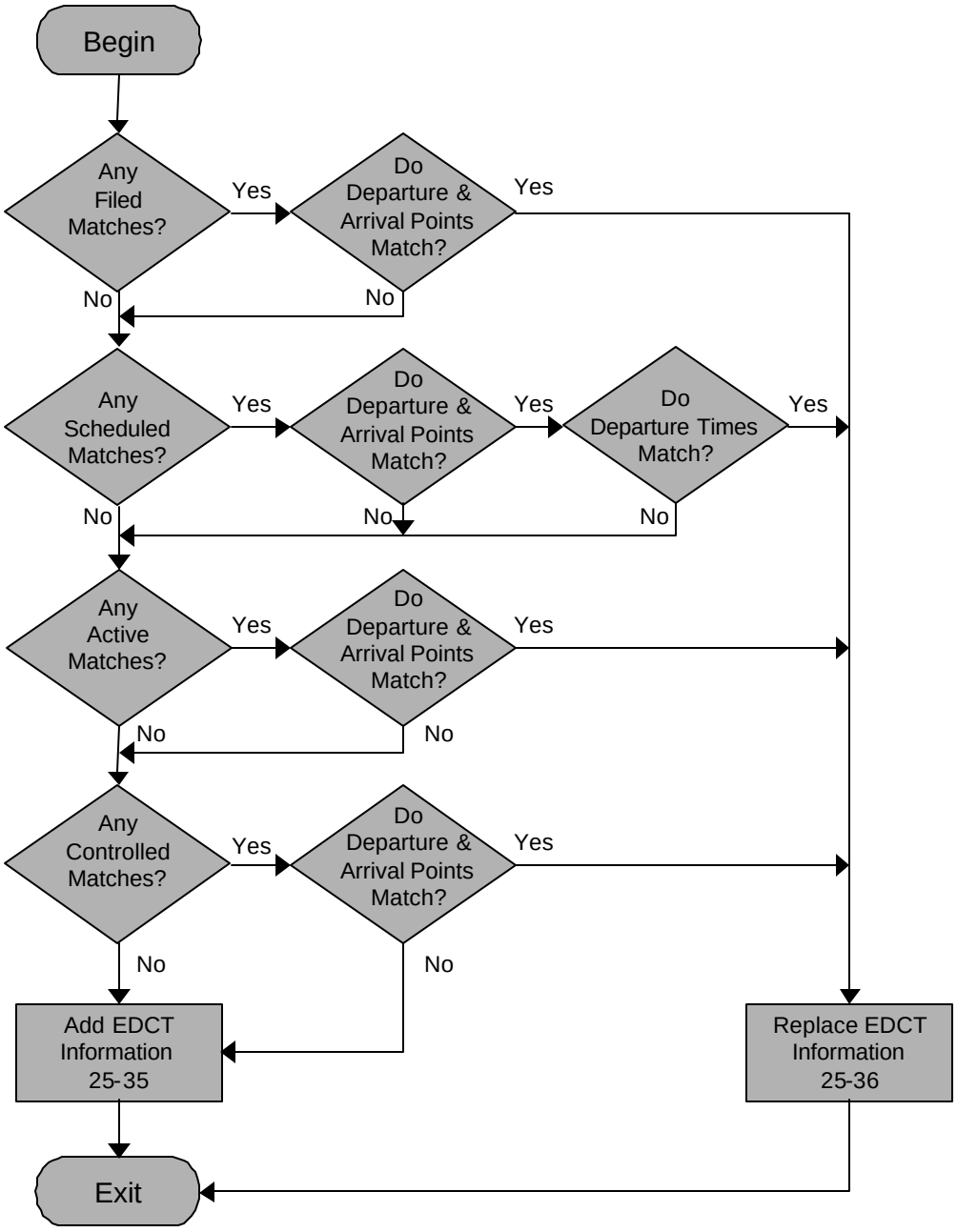

Figure 25-34. Sequential Logic for the Do_EDCT Module

Logic which creates a new flight entry in the FDB with the EDCT message information is shown in Figure 25-35. *Do_EDCT* first allocates a new flight record, and inserts the flight into FDB. The update databases flag is set to TRUE, and the TDB update type is set to add.

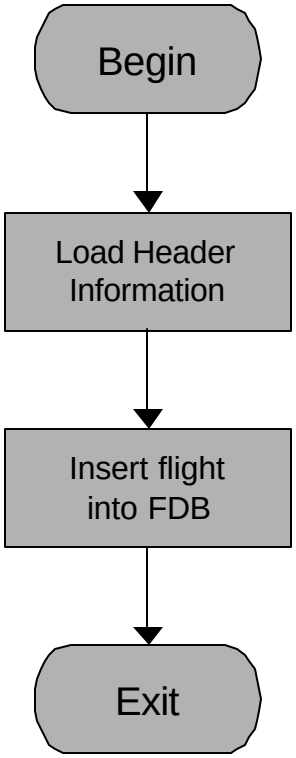

Figure 25-35. Sequential Logic for the Add EDCT Flight Information Routine

Do_EDCT updates an existing FDB entry with EDCT message information as shown in Figure 25-36. *Do_EDCT* replaces the header record with the information contained in the EDCT message. If the matching entry has an event list and the first event is a departure event, *Do_EDCT* updates the first event time, computes flight times for the rest of the event list, and replaces the event list in the **evdb**. *Do_EDCT* updates the time arrays, sets the update databases flag to TRUE and sets the update type to replace.

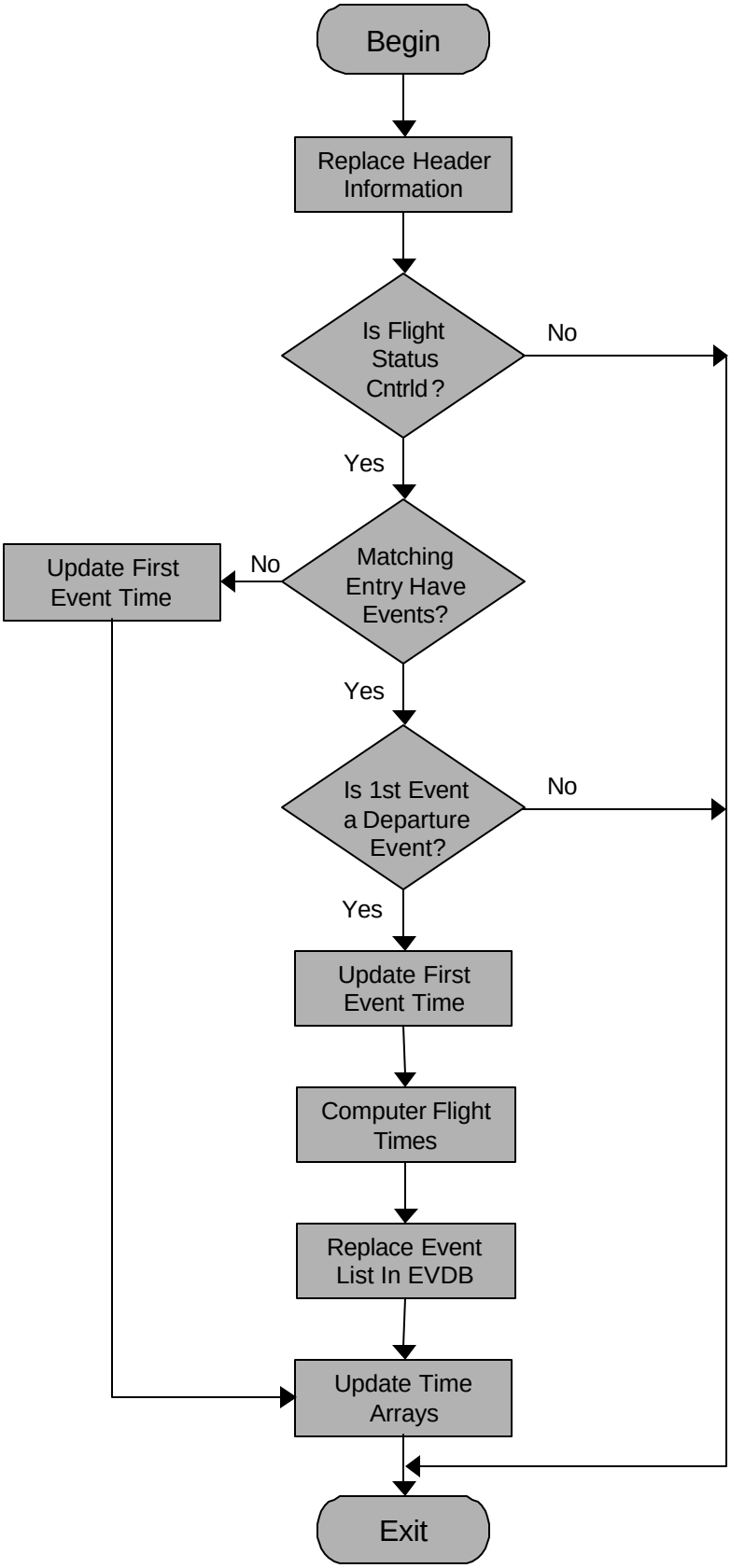

Figure 25-36. Sequential Logic for the Replace EDCT Flight Information Routine

25.14.4.5.4 The Do_DZ Module

DZ (departure) messages received by *Process Flight Messages* are processed by the *Do_DZ* module as shown in Figure 25-37. *Process Flight Messages* first checks the message's flight ID against the hash table (see Section 25.14.4). If no matching flights are located, the DZ message information is added to the database. Any matching flights are sorted by flight status before applying the DZ-specific message matching criteria.

Do_DZ first checks for computer ID matches, then for matches with **filed, controlled**, **scheduled**, and **active** flights. A computer ID match or a match with a **controlled** or **scheduled** entry causes a check for any currently active flights with the same flight ID before updating matching database entry with the information contained in the DZ message. For messages originating outside of CONUS, an attempt is made to match with an active flight. To be considered a match, the arrival airports must be the same and the coordination fix time in the DZ message must be earlier than the last TZ time. A duplicate flight plan condition occurs if a filed flight entry matches the message in every way except the computer ID.

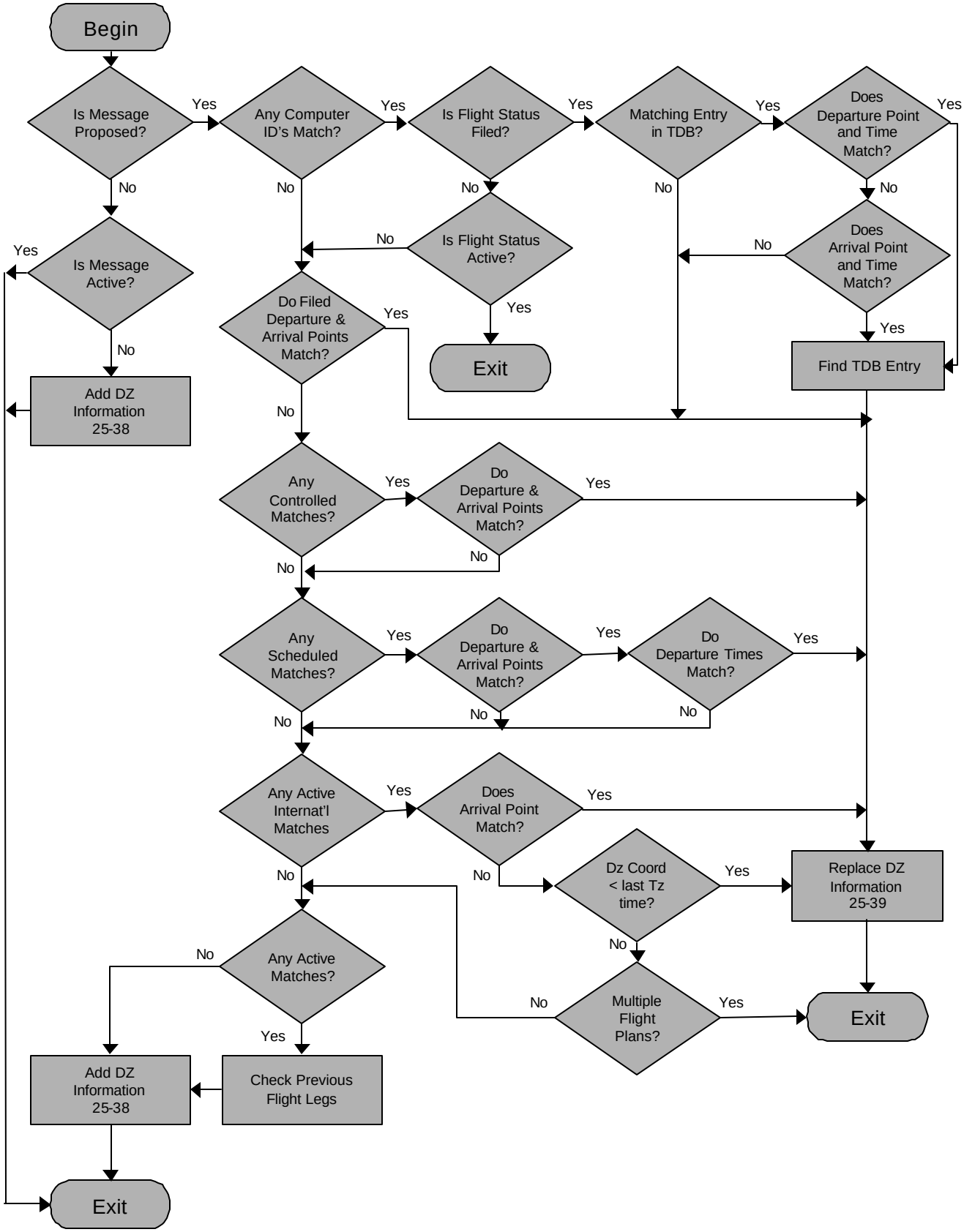

Figure 25-37. Sequential Logic for the Do_DZ Module

In order to add a DZ message to the database (see Figure 25-38), *Do_DZ* first allocates a new flight record and inserts the message information. Next, it checks the message's coordination fix to see if it corresponds to an airport departure event. If it does, *Do_DZ* then adds the departure event to the **evdb**. The update databases flag is set to TRUE, and the TDB update type is set to add.

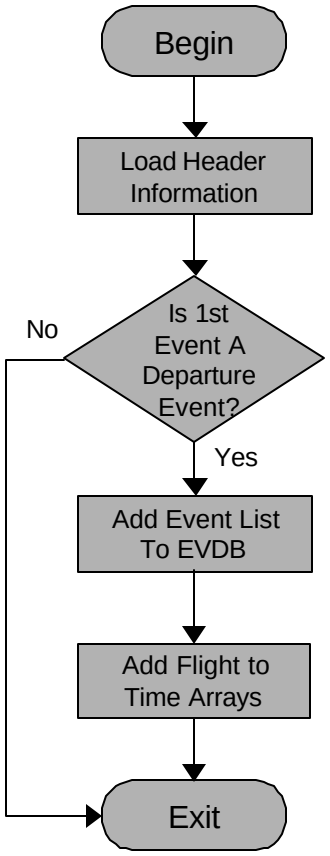

Figure 25-38. Sequential Logic for the Add DZ Flight Information Routine

Updating an existing flight record (see Figure 25-39) begins when *Do_DZ* replaces the record header information with the data contained in the message. If the matching entry has an event list, *Do_DZ* updates the entry's event list under one of three conditions:

(1) The coordination fix event in the message matches the first event in the entry's event list.

(2) The coordination time in the DZ message is an estimated (E) time.

(3) A proposed flight plan containing a tailored route was received for this flight.

If none of these conditions are met, *Do_DZ* logs an error and returns without processing the message. Once the first event time has been updated, *Do_DZ* recomputes the flight times in the event list and replaces the event list in the EVDB. If the matching entry does not already have an event list, and if the coordination fix event in the message is a departure airport event, *Do_DZ* adds the event to the **evdb**. The update databases flag is set to TRUE and the TDB update type is set to replace.

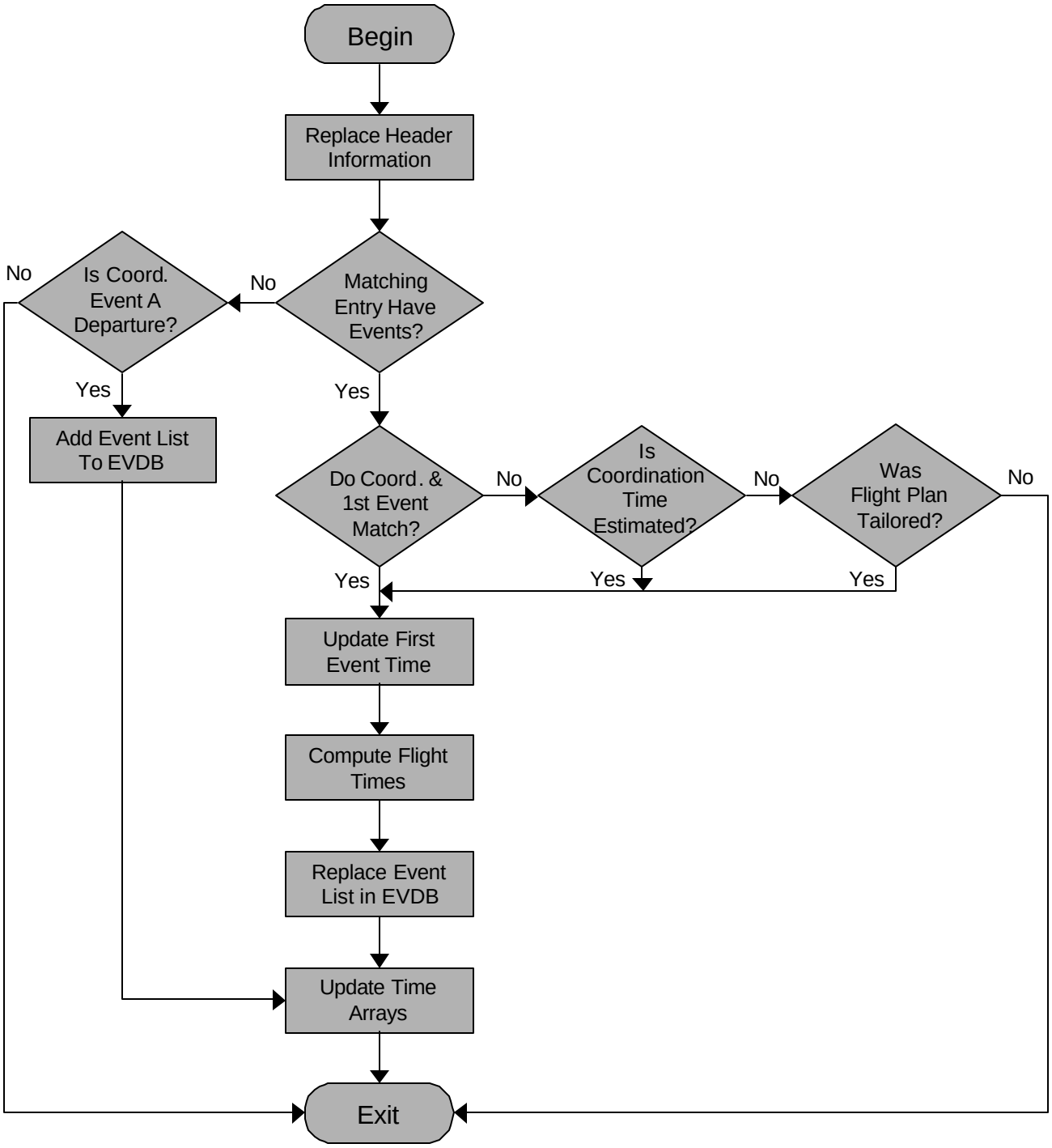

Figure 25-39. Sequential Logic for the Replace DZ Flight Information Routine

25.14.4.5.5 The Do_UZ Module

Process Flight Messages utilizes the *Do_UZ* module to perform processing of UZ (boundary crossing) messages as shown in Figure 25-40. As in other message processing, *Do_UZ* accesses a list of addresses which have been sorted by flight status during message matching. For active messages, *Do_UZ* gives highest matching priority to **active** flights.

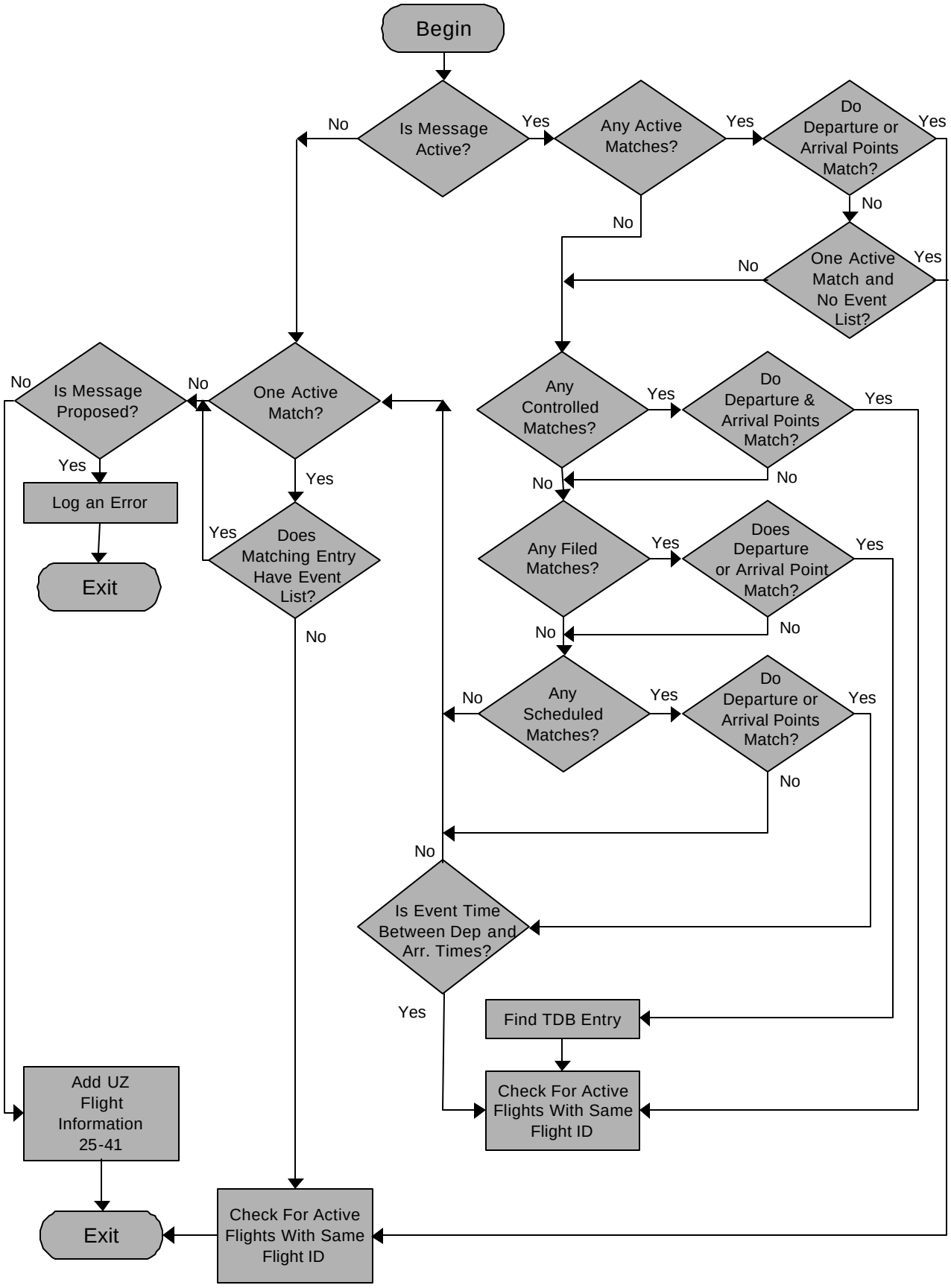

Figure 25-40. Sequential Logic for the Do_UZ Module

After **active**, *Do_UZ* checks for **controlled**, **filed**, and **scheduled** matches. If exact matches for an active message fails, *Do_UZ* checks for a single matching entry that does not have an event list. Otherwise, when a proposed message is received, *Do_UZ* logs an error and discards the message.

In order to add the information from a UZ message to the database (see Figure 25-41), *Do_UZ* first moves the information from the message into a newly allocated flight record.

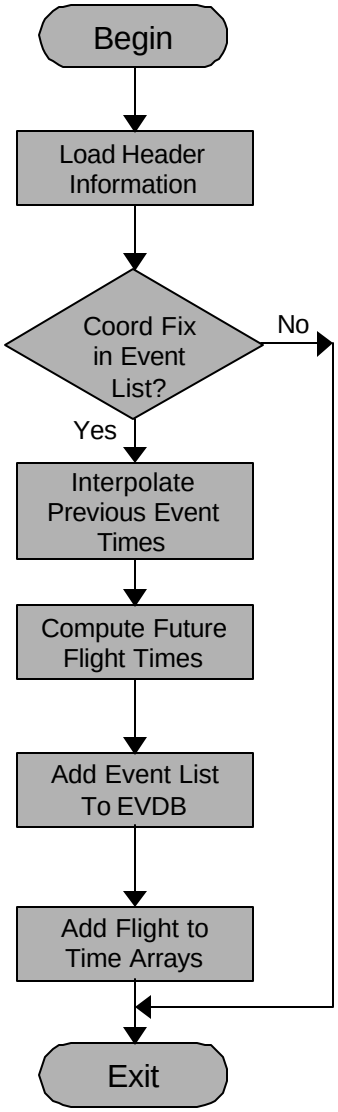

Figure 25-41. Sequential Logic for the Add UZ Flight Information Routine

Next, *Do_UZ* compares the message's coordination fix to the event list in order to find the event which most closely matches. It interpolates flight event times previous to the matching event and models flight times over the future portion of the event list. The update databases flag is set to TRUE, and the TDB update type is set to add.

If a matching entry is found and that flight has an event list, *Do_UZ* follows the logical flow depicted in Figure 25-42 to update the matching entry.

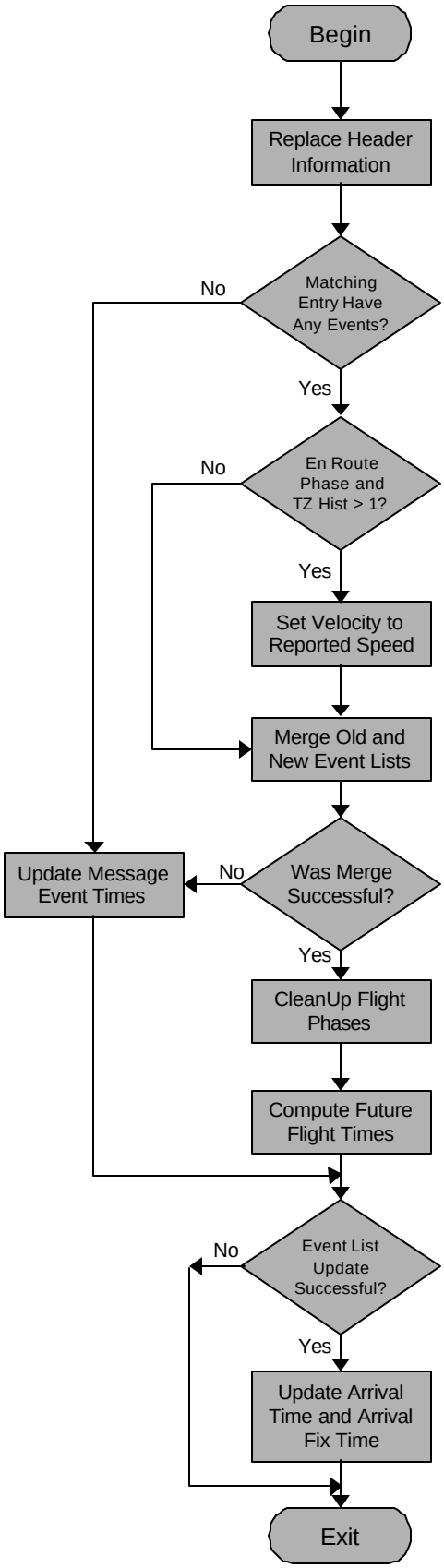

Figure 25-42. Sequential Logic for the Replace UZ Flight Information Routine

Do_UZ first attempts to merge the old and new event lists. If the merge is successful, *Do_UZ* computes future flight times in the merged list based on the message's coordination time. If the merge is unsuccessful, *Do_UZ* replaces the old event list with the new list, interpolates flight times for events previous to the coordination fix, and computes any future flight event times. The update databases flag is set to TRUE, and the TDB update type is set to replace.

25.14.4.5.6 The Do_TZ Module

The message matching sequence for TZ (position update) messages is shown in Figure 25-43.

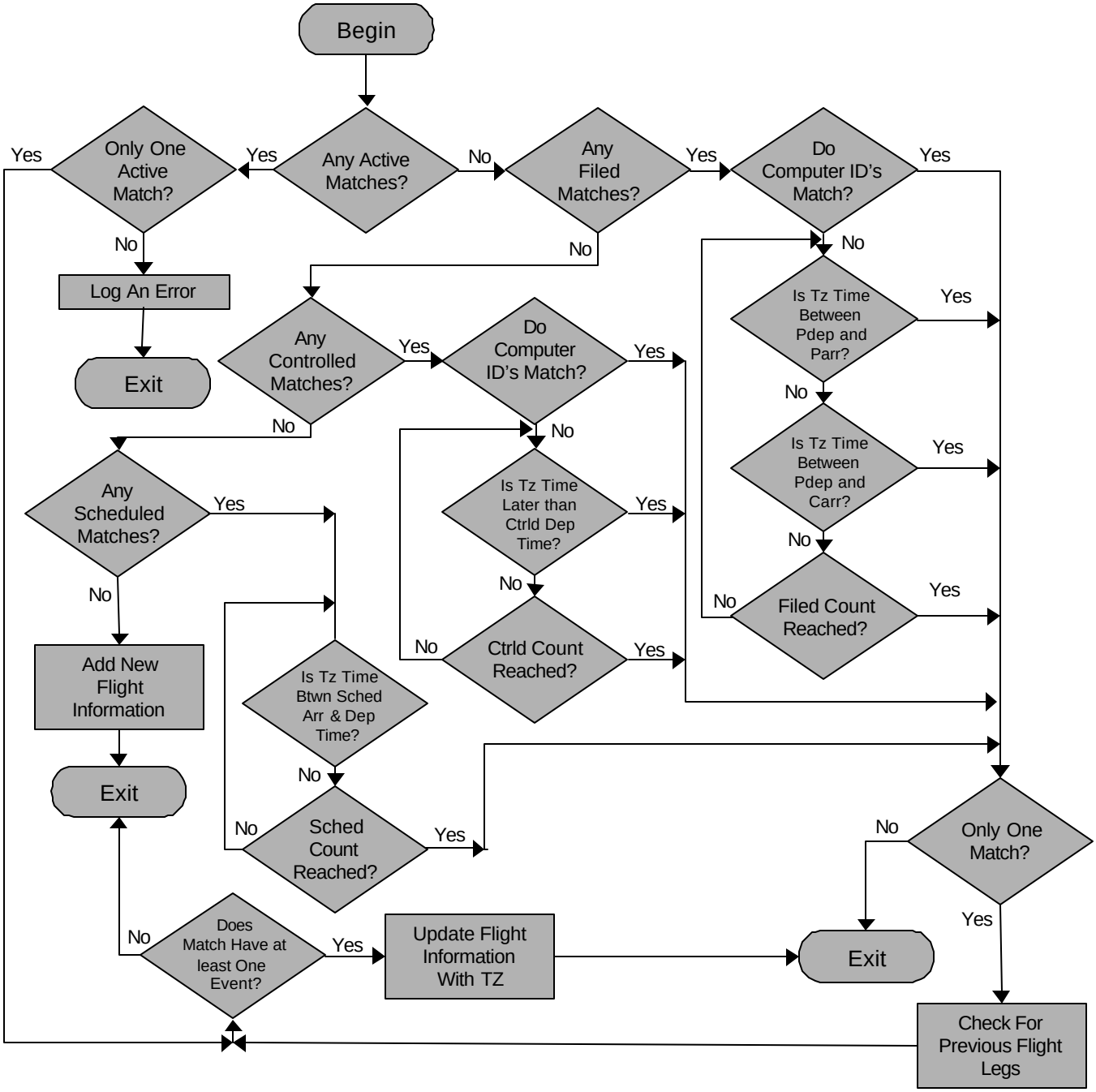

Figure 25-43. Sequential Logic for the Do_TZ Module

Due to the limited amount of information contained in TZ messages, the message is ignored if no flight ID match is found. When a flight ID match is found, highest priority is given to active flights followed by filed and controlled flights. An active match is only allowed if there is only one active flight with the same flight ID as the message. When *Do_TZ* locates a redundant active flight, it logs an error and discards the message. For controlled and filed flights, the matching entry's computer ID is also checked to ensure an accurate match. If this computer ID check is successful, *Do_TZ* first checks for and deactivates any active flights with the same flight ID as the current message. In all cases, *Do_TZ* checks to make sure that the event list contains at least two events before proceeding with position update processing, which is done by the *TZProcessingForFDB* module.

25.14.4.5.7 The TZProcessingForFDB Module

The *TZProcessingForFDB* module updates the matching flight's Flight Database entry with information from a parsed position update (TZ) message. A TZ message contains the latest actual flight location, velocity, and altitude; *TZProcessingForFDB* applies this real-time, real-life information to the flight's predicted path (the event list) to best estimate the velocity, altitude, and, most importantly, time of the flight at each point (event) in the path.

In general, once the appropriate FDB entry has been found for the input TZ message, *TZProcessingForFDB* updates the flight's entry as shown in the flow chart of Figure 25-44. This section will describe this higher level processing; the important subroutines, *UpdateEventListForTZ, VerifyFlightOnRoute, and NextPositionPrediction*, and *RecoverLastActualEventBy TZLocation* will be described in future sections.

TZProcessingForFDB first checks the parsed TZ message for valid data (e.g., speed should be non-zero). If the flight had been determined to be taking its en route delay (as assigned in the flight plan), *TZProcessingForFDB* checks further whether the time on the TZ message is later than the predicted ending time of the en route delay. If the TZ time is earlier, some information from the message is saved in the FDB, but basically, the message is ignored. If the TZ time is later, *TZProcessingForFDB* continues with its normal processing.

> **NOTE:** The terms location status and last actual event are introduced below, but are described in greater detail in the description of the *RecoverLastActualEventByTZLocation* routine.

After retrieving the flight's event list, *TZProcessingForFDB* performs five of seven major checks in order to verify TZ information completeness and consistency with the flight record and event list:

(1) If the first event in the event list corresponds to the departure airport, make sure that the current value of the last actual event stored in the flight record is at least equal to one.

(2) If this is the first TZ message or if a route change has occurred, check to make sure that the event list is consistent with respect to phase, altitude, velocity.

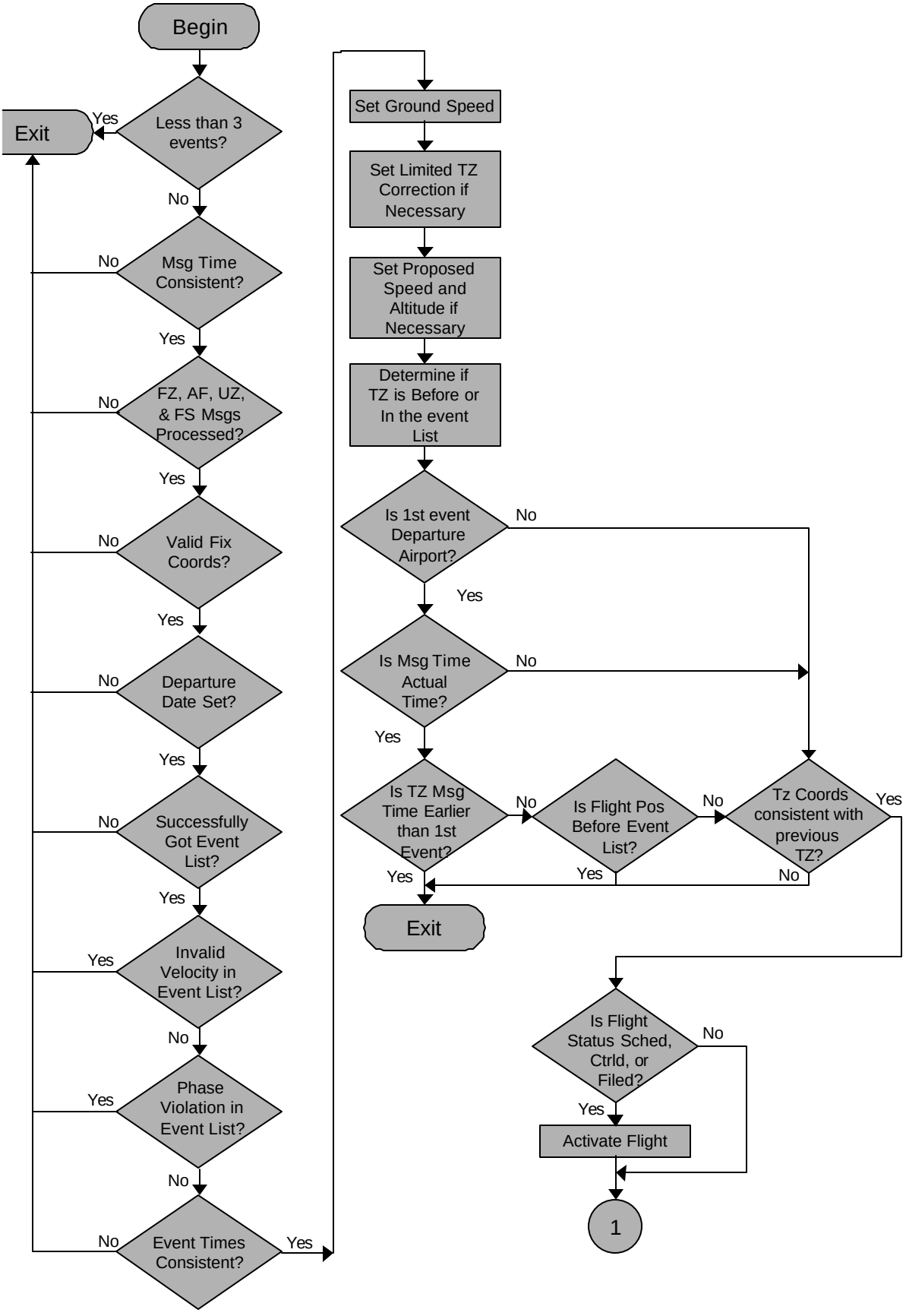

Figure 25-44. Sequential Logic for the TZProcessingForFDB Process

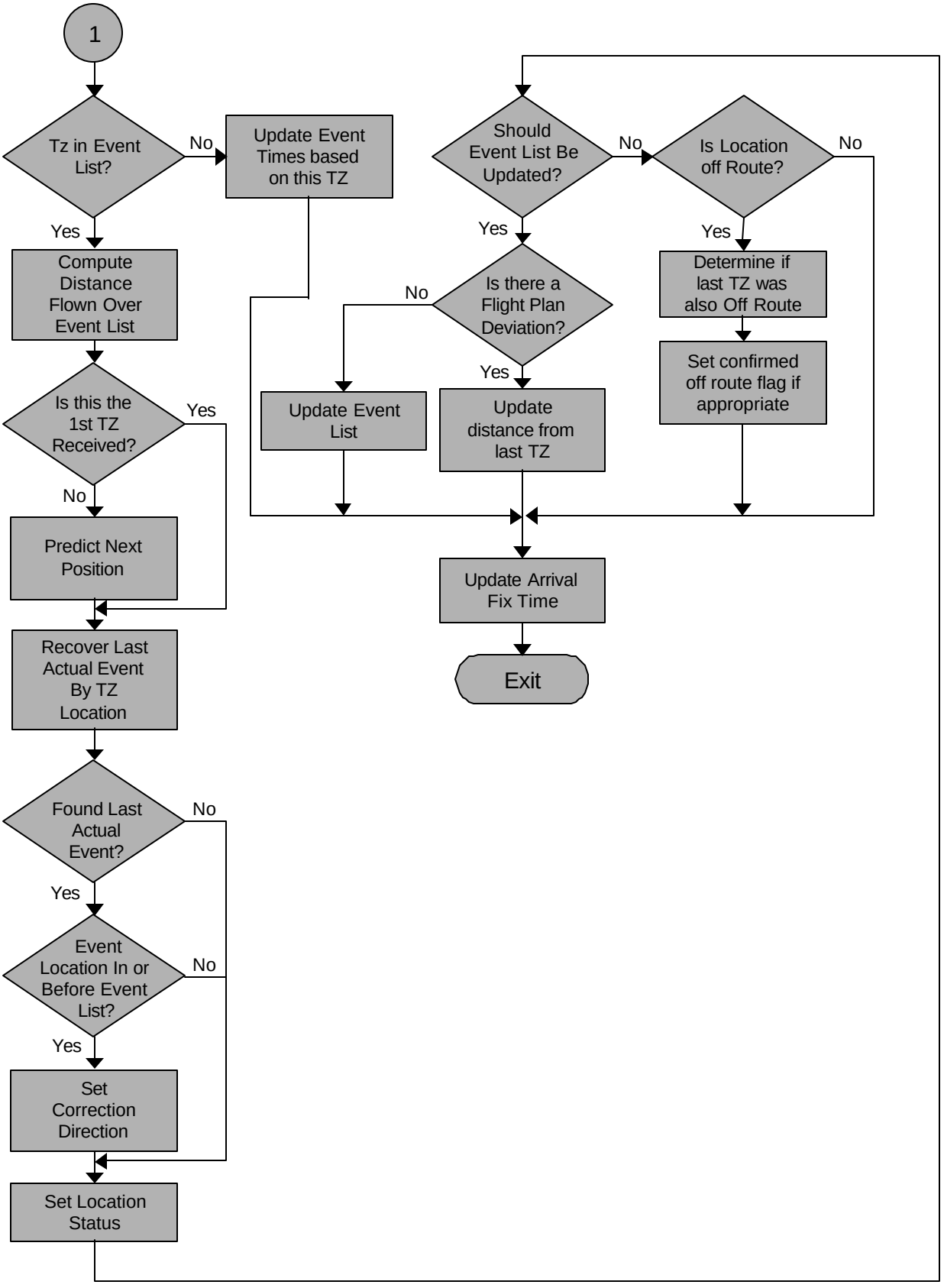

Figure 25-44. Sequential Logic for the TZProcessingForFDB Process (continued)

(3) If the ground speed in TZ message is missing, attempt to find a replacement from past TZ messages or the flight plan. If no value is available, a limited TZ correction is made based solely upon the TZ location.

(4) The ground speed obtained from the message or during check (3), is compared to the speed in the last actual event or the filed speed. Limited TZ correction will also result if the the new speed is found to be unreasonable.

(5) If TZ message has a valid speed, but the proposed speed and/or proposed altitude are not set for the flight, they must also be set according to the best information available.

Before performing the final two major checks, *TZProcessingForFDB* retrieves the flight's previous location status. This status can be one of two values, **before** or **in**, depending on the flight's position (as described by the TZ message) with respect to the event list. The current location status is determined later in *TZProcessingForFDB*'s processing by the *RecoverLastActualEventByTZLocation* routine.

With the previous location status, *TZProcessingForFDB* performs the following two final checks:

(1) If the first event in the event list is the departure airport and contains an actual departure time, make sure that the TZ time is after this departure time. Also verify that for this case the TZ is **in** the event list.

(2) This is the final check for possible corruption of coordinates. TZ coordinates must be consistent with the previous TZ coordinates saved in FDB. This means the geographical distance between TZs has to be close to distance computed with Isaac Newton's method, i.e., speed multiplied by time difference.

If current TZ position is **before** the event list or if this is the first TZ for the flight and the actual departure time (from a DZ) is missing, *TZProcessingForFDB* updates the time for all events in the list, beginning with the first, based on the best available ground speed and the distance between the TZ position and the first event.

If previous TZ is **in** the event list, the routine *NextPositionPrediction* is called. This routine attempts to predict the current flight position with respect to the last TZ message. If this prediction fails, or if it compares poorly with the actual position reported in the current TZ message, *TZProcessingForFDB* does not update the event list times. However, if after all checks have been completed, the current TZ location is found to be **in** the event list, *TZProcessingForFDB* invokes the routine *RecoverLastActualEventByTZLocation* to find the new last actual event and determine location status.

When the TZ location is found to be **in** the event list, *RecoverLastActualEventByTZLocation* seeks the the location of the event in the event list which is previous and closest to the position depicted in the current TZ message. This event is called the last actual event. Using this last actual event, *TZProcessingForFDB* can interpolate times for past events and predict times for future events.

The location status is used throughout the TZ message processing; it is determined by the following three questions:

(1) Is TZ location within the *limit distance* from the flight path?

(2) Is there is a delayed event right after TZ location?

(3) Is the TZ location **before** or **in** the flight event list?

The concepts of **before** and **in** must be considered, because each event list does not necessarily begin at the departure airport (e.g., when the event list was obtained from a UZ message). This means that part of the event list is missing, and that TZ messages are arriving that correspond to some point within the missing portion. The actual location for this case is considered **before** the geographic point where the event list actually begins. If the TZ message position corresponds to a point after the first event, the message is considered to be **in** the event list.

The *limit distance* is the maximum distance allowed between a flight's current position (from the TZ) and its predicted route of flight (the event list) before a flight is considered off-route. A delayed event is an event for which a delay value was specified in the initial field 10. The TZ location is determined to be **before** or **in** the event list depending on its position relative to the first event in the event list.

RecoverLastActualEventByTZLocation returns the following values of location status:

- 0 when the flight is near the route, no delayed event found.
- 1 when the flight is off the route, no delayed event found.
- 2 when the flight is near the route and in delayed event area.
- 3 when the flight is off the route and in delayed event area.
- 4 when TZ time shows a time earlier than the time for the new last actual event and when the TZ position is near an event with a filed delay.
- 5 when TZ location is determined as being **before** the event list.

Values 0 –– 4 correspond to a TZ which is **in** the event list. If location status = 0, then *TZProcessingForFDB* invokes the routine *UpdateEventListForTZ* (Section 25.14.4.5.8) which updates events using the new time, ground speed, and altitude from the TZ. If location status = 1 then *TZProcessingForFDB* invokes the routine *VerifyFlightOnRoute* (Section 25.14.4.5.10), which determines whether the flight is currently on or off its proposed route. If the flight is determined to be on its route, *TZProcessingForFDB* invokes *UpdateEventListForTZ*. Location status values of 2, 3, or 4 causes the initiation of a delay period that will be used during subsequent TZ processing. During this delay period, TZs will not be used to update event list times. As has been outlined previously, a location status of 5 (TZ is **before** the event list) causes only future event times to be computed.

Next, *TZProcessingForFDB* replaces any event list changes in the EVDB. *TZProcessingForFDB* then updates the appropriate fields in the FDB record (current airport arrival time, arrival fix time). The update databases flag is set to TRUE, and the TDB update type is set to replace.

TZProcessingForFDB may encounter numerous error conditions during processing of TZ messages. Some of these error conditions are provoked by live data corruption: any part of the TZ message may contain an error or be missing. The other major source of error is the irregular sequence of the messages themselves, particularly when a flight path correction is made.

TZProcessingForFDB attempts, wherever possible, to correct any error it detects. If such correction is not possible, *TZProcessingForFDB* will immediately cease processing of the erroneous message, which presents corruption of current flight information. Errors which indicate database corruption or numerical inconsistencies between a TZ message and the flight information currently stored in the FDB will generate an error message on the main *fdb_manager* transcript pad.

25.14.4.5.8 The UpdateEventListForTZ Module

The *UpdateEventListForTZ* module uses information from the parsed TZ message to update the time, speed, and altitude values in certain events of a flight's event list. *UpdateEventlistForTZ* also uses certain control information as determined by *TZProcessingForFDB*. This information determines how to select the event at which the update will begin and whether *UpdateEventlistForTZ* should use the TZ supplied speed for this update. Upon successful completion, *UpdateEventListForTZ* returns an updated event list.

The logical flow of *UpdateEventListForTZ* is shown in Figure 25-45. *UpdateEventListForTZ* begins by comparing the TZ supplied speed and altitude to some reasonable values. If the speed and altitude do not fall within these limits, the module will use the proposed cruising speed and altitude in its predictions.

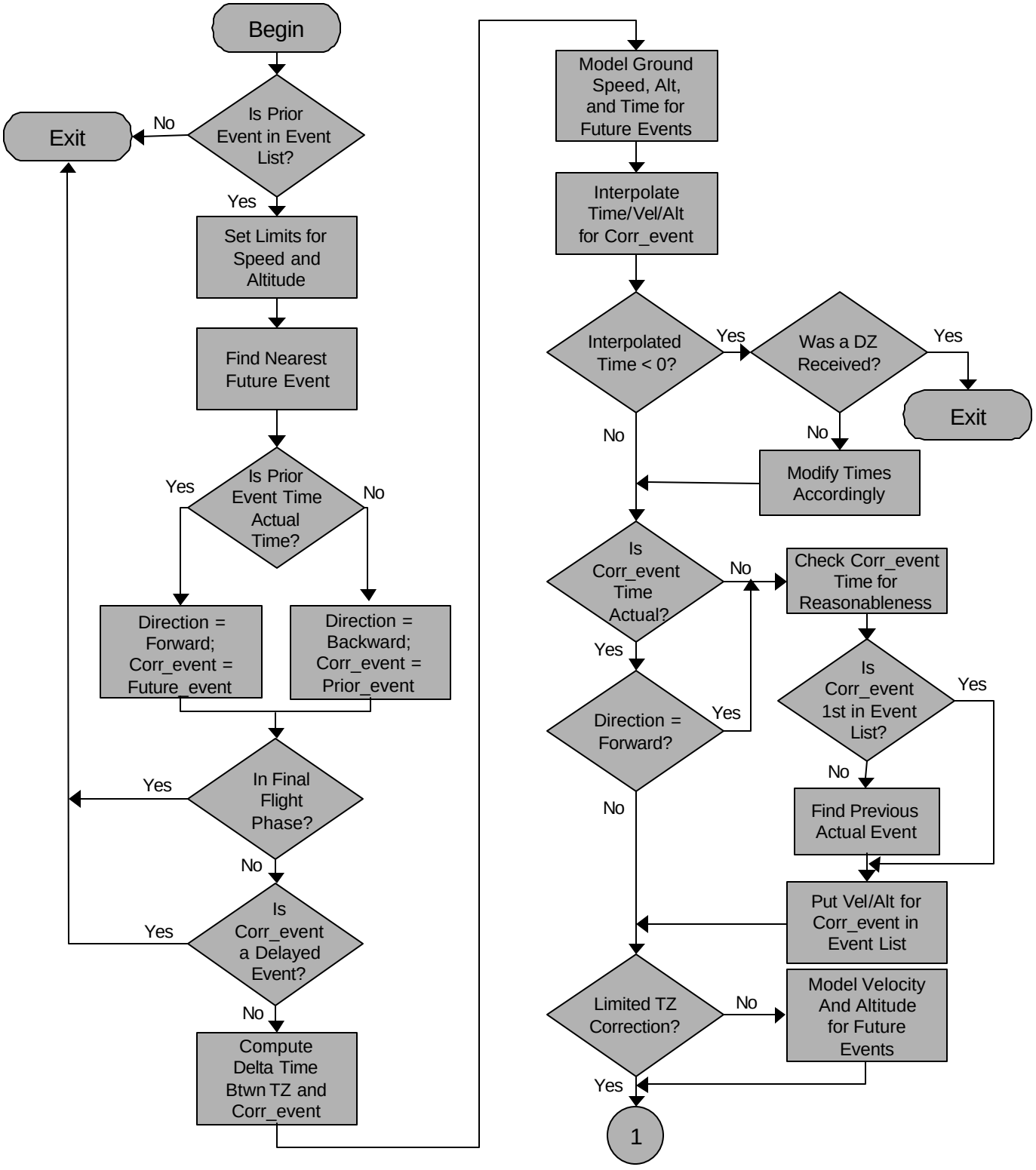

Figure 25-45. Sequential Logic for the UpdateEventListForTZModule

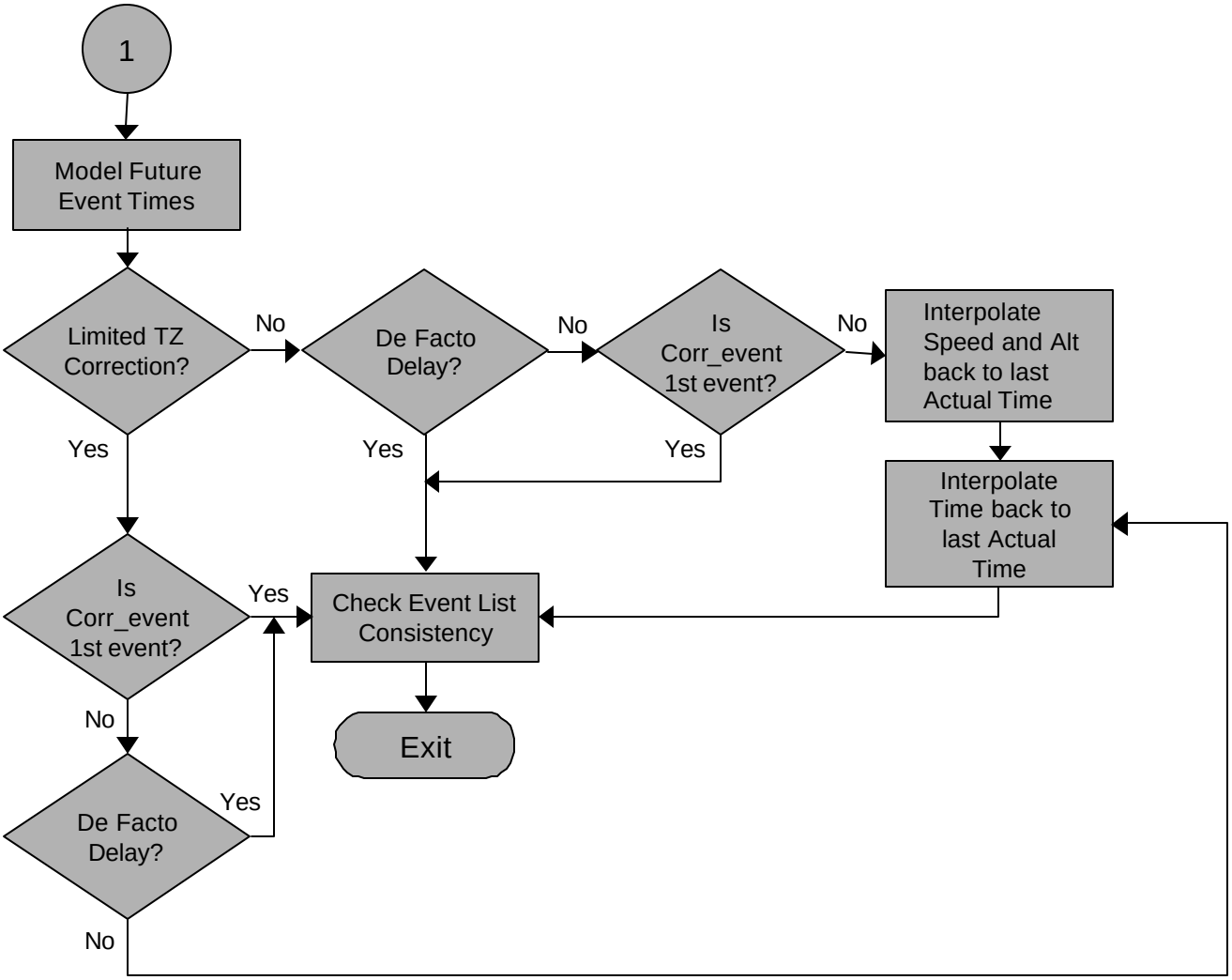

Figure 25-45. Sequential Logic for the UpdateEventListForTZModule (continued)

Update*EventListForTZ* then uses the supplied event list and last actual event value to determine the direction of the time update. The direction is assumed to be BACKWARD, by default. If the last actual event has already been updated by actual information from a previous TZ, the direction is set to FORWARD.

Using the direction of update, *UpdateEventListForTZ* selects an event in the list as the *correction event*. This correction event is the event most closely associated with the position from the TZ. Usually, the direction of update is **BACKWARD** and the last actual event is selected as the correction event. However, if the update direction is **FORWARD**, the last actual event was the correction event already; so the closest future event will be used instead. This correction event is used as an anchor point; *UpdateEventListForTZ* interpolates times for events before this event in the list and models times for events after it. No correction is done if correction event is the last one in the event list or if its phase value is **approach** or **landed**.

UpdateEventListForTZ uses the position from the correction event and the flight's position from the TZ to compute a distance between them. Using this distance and the speed of the flight, *UpdateEventListForTZ* computes a delta time. If the update direction is **BACKWARD**, the new time at the correction event is equal to the the TZ time stamp time minus the delta time. If the update direction is **FORWARD**, the TZ time stamp time plus the delta time equals the correction event's new time.

Make updates for speed and altitude for future events, i.e., *model* them: if they are not on en route phase make an extrapolation for speed and altitude until en route phase met or the altitude reaches cruising level. They are considered to be on en route phase of flight, if the last actual event is on en route phase of flight, and the altitude obtained from TZ is not more than 5% less than proposed altitude. The adjustment for events velocities in arrival, approach, and landing phases are made by a simple rule: do not allow them to exceed cruising velocity. After these changes, the events' times correction can be done. The time correction will not be performed for delayed events. Now, the same type of correction is done for the past events, i.e., *interpolation*, before the first event with actual information from the past TZ is met.

During *UpdateEventListForTZ*'s processing, certain circumstances may cause errors resulting in logical inconsistencies in the event list. These inconsistencies are listed below; when they occur, *UpdateEventListForTZ* signals the calling routine so that the event list is not replaced in FDB:

- Neither the TZ nor the correction event has a valid speed and therefore even the limited correction cannot be done.
- *InterpolTimeVelocityAltitude* failed to interpolate values for correction event from TZ information.
- The new correction event time is beyond the reasonable limit: <=0 or >2880.
- Events phases are in wrong order in the list.

25.14.4.5.9 The NextPositionPrediction Module

The *NextPositionPrediction* module uses information from the parsed TZ message, the matching flight's event list, and the regular time interval between successive TZs for a particular flight (currently five minutes), to compute the spherical coordinates (latitude , longitude) for the next expected plane position on the route (as determined by the event list). These predicted coordinates are used by *TZProcessingForFDB* to help check the reasonableness of information from the next parsed TZ message.

The logical flow of the *NextPositionPrediction* module is shown in Figure 25-46. *NextPositionPrediction* first computes the difference between the time stamp from the previous TZ and the time at the previous last actual event. This time difference is added to the regular TZ interval, and *NextPositionPrediction* uses this total time difference and the speed from the last actual event to compute a relative distance flown since that event. Using this predicted relative distance and distances between events in the event list, *NextPositionPrediction* finds the closest likely event at the next TZ time. Finally the subroutine *InterpolCoordinatesCompute* calculates spherical coordinates for predicted flight position using coordinates of the two closest events on the route and a distance at which the third point shifted off the first event.

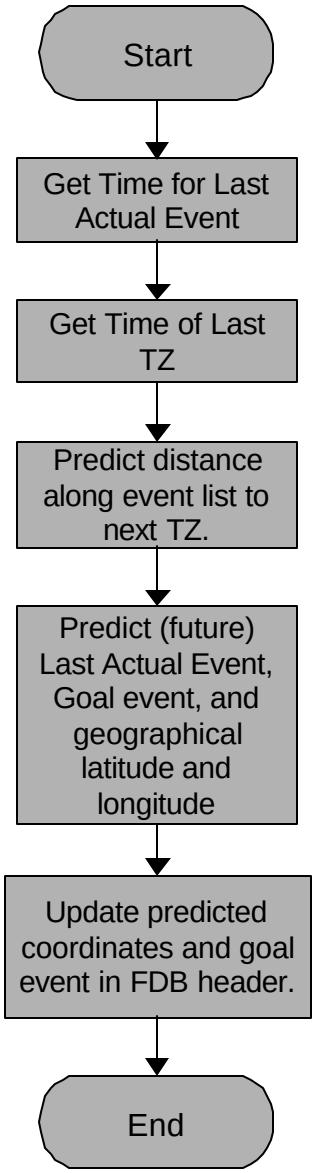

Figure 25-46. Sequential Logic for the NextPositionPrediction Module

All *NextPositionPrediction* processing assumes that the flight's position, as described by TZ messages, is close (within some tolerance) to the flight path defined by the event list. If the flight's position is not, *NextPositionPrediction* fails, and dummy predicted coordinates are set.

25.14.4.5.10 The VerifyFlightOnRoute Module

The *VerifyFlightOnRoute* module checks if the flight continues along the planned route (as determined by the event list) or if it diverges for any reason. The verification is done only after *TZProcessingForFDB* determines that the shortest distance between TZ location and the route exceeds a deviation limit provided in the *FDP* start-up parameters.

The *VerifyFlightOnRoute* module uses the following input in its determination of the flight's position status:

(1) TZ message information.

(2) Flight's actual shortest distance from the route at the time.

(3) A deviation limit measured in miles.

(4) A deviation slope limit given as a constant in miles per minute and the weight coefficient alpha used for an exponential filter.

Figure 25-47 represents Sequential Logic for *VerifyFlightOnRoute.*

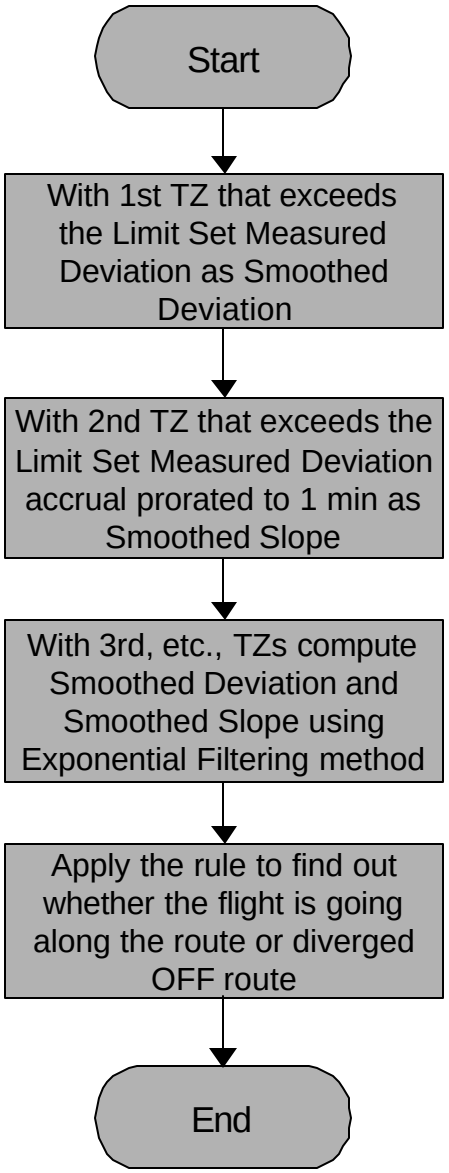

Figure 25-47. Sequential Logic for the VerifyFlightOnRoute Module

An exponential smoothing recurrent method (exponential filter) is used for measuring deviation of the flight from its route. The deviation and its time derivative are substituted into recurrent formulas to compute an estimation of average deviation and a derivative of deviation as a function of time. This last value is the slope of deviation and is a function of time itself. Physically, this is a plane speed in the off-route direction. This plane speed is measured in miles per minute to make it independent of the between-TZ-time interval, which actually varies unpredictably. The following global parameters are used:

(1) **MAXALLOWEDDEV** – Maximum allowed deviation from the route (miles).

(2) **MAXALLOWEDDEVSLOPE** – Maximum allowed deviation slope from the route (miles per minute).

(3) **WEIGHTCOEFFICIENT** – Weight coefficient for the exponential filter (0-1).

Upon completion, the *VerifyFlightOnRoute* module returns TRUE if (at this time) the flight still may be considered as following the route; otherwise, it returns FALSE.

25.14.4.5.11 The Do_TO Module

The message-matching sequence for TO messages (oceanic position updates) is shown in figure 25-48. The first step is to convert the TO position time into Julian minute format. This is done to simplify subsequent time comparisons.

Next, *Do_TO* searches the provided list of FDB flight entries for the best match. The manner of the search is shown in figure 25-49. *Do_TO* prioritizes the search based on flight status. It searches for active flights first, followed by scheduled, filed, and controlled. For scheduled and filed flights, *Do_TO* imposes the additional restriction that the time contained in the TO message must fall within the departure/arrival interval of the flight.

Further processing is dependent on the number and type of flight matches found. If no FDB flight entries match the TO message, no action is taken and *Do_TO* returns TRUE. If there are multiple matches, *Do_TO* returns FALSE and for multiple active matches, prints an error message. If *Do_TO* finds a single match, then processing continues.

If the single matching flight is active, *Do_TO* determines whether a TZ message was received within a preceding time out interval. If so, the TO message is discarded and *Do_TO* returns *true*. This is done so that more accurate TZ messages will take precedence over TO messages. In all other cases, *Do_TO* calls *TOMprocessingforfdb* to update the FDB flight record with the TO flight data. Finally, *Do_TO* sets its return value based on the success of the update.

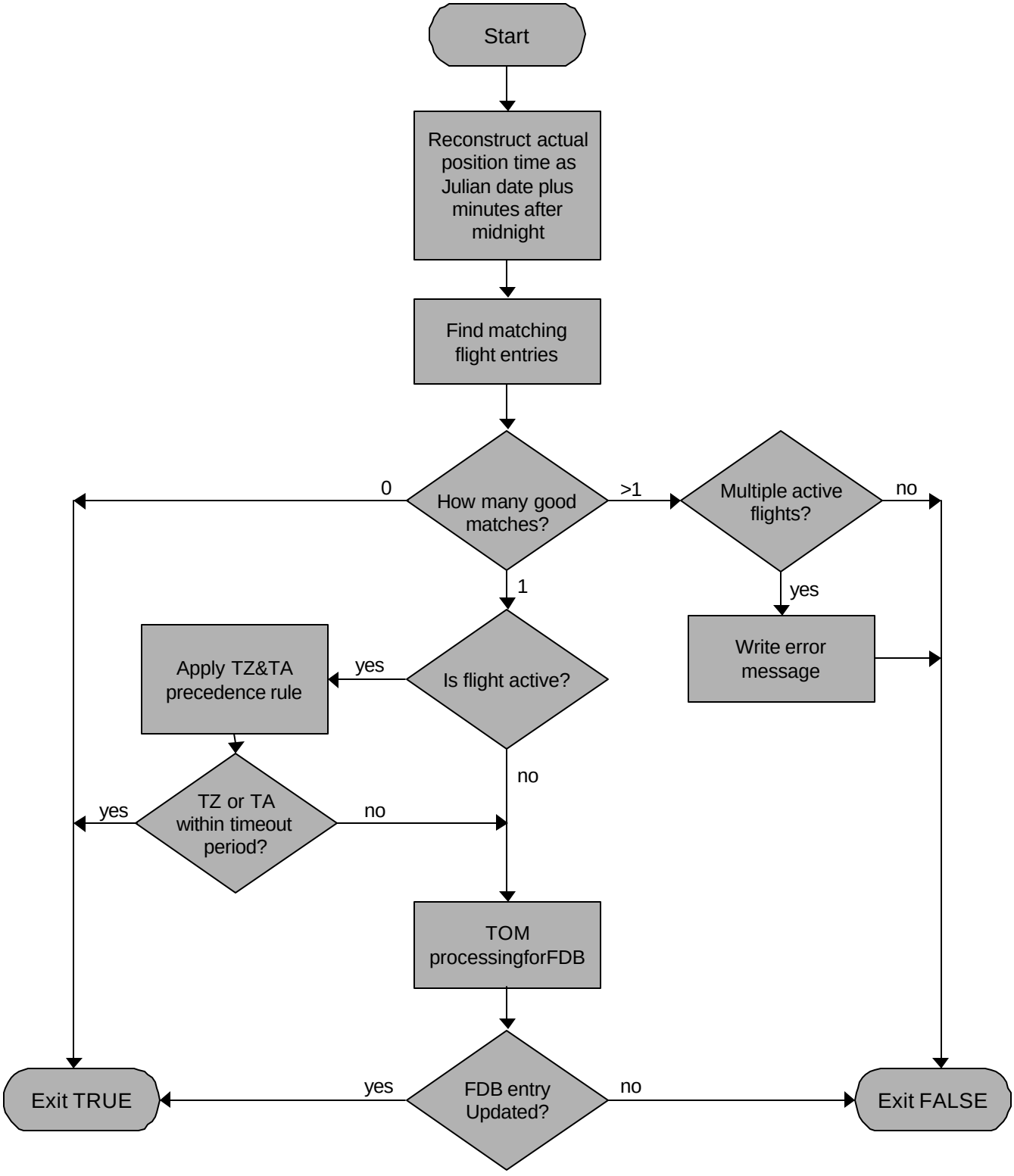

Figure 25-48. Sequential Logic for the Do_TO Module

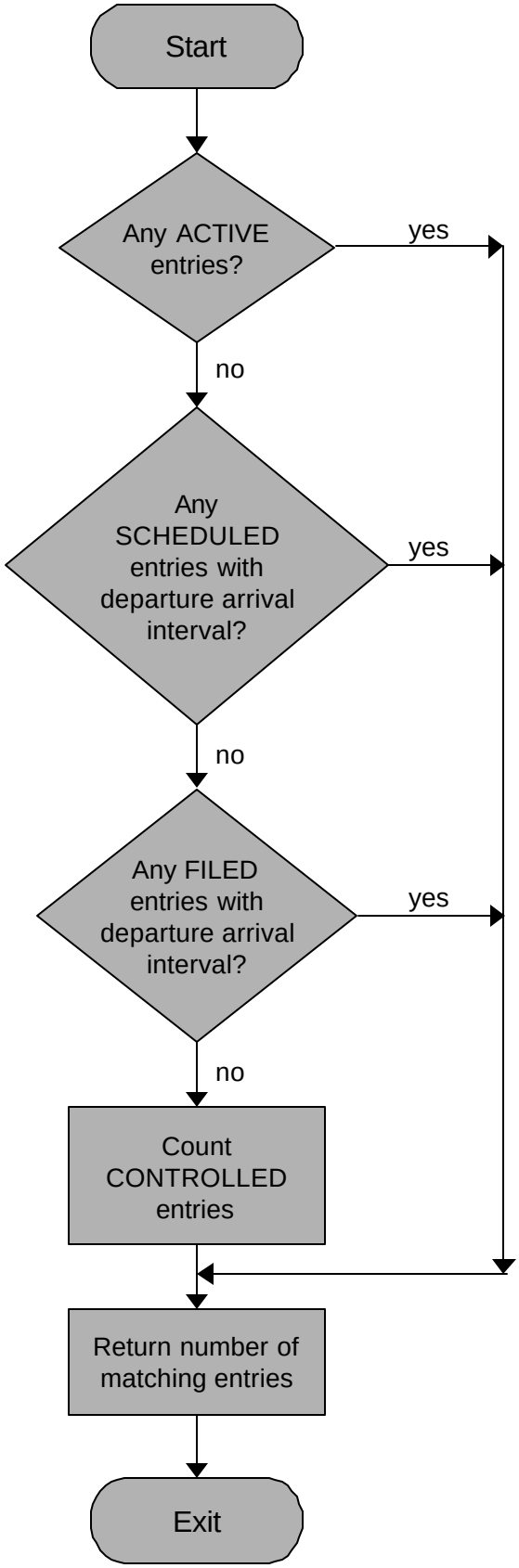

Figure 25-49. Sequential Logic for Find Matching Flight Entries Routine

25.14.4.5.12 The TOMprocessingforfdb Module

The *TOMprocessingforfdb* module updates the matching Flight Database entry with information from the TO message. *TOMprocessingforfdb* updates the FDB flight record by performing the steps depicted in Figure 25-50. The module saves the existing FDB flight record, updates a copy of the record, and if updated successfully, replaces the saved flight record with the updated record.

TOMprocessingforfdb sets the flight status to active and performs validity checks on each component of the TO message. The module verifies that the coordinates, reported ground speed, altitude, and time fall within appropriate ranges. If not, the routine returns FALSE.

Next, *TOMprocessingforfdb* locates the flight position within the flight path, i.e., it finds the last actual event flown by the aircraft. The position of the last actual event along with the flight's deviation from the route determine whether it is possible to update the eventlist with the TO data.

It is not possible to update the eventlist when the TO flight position occurs prior to the first event in the eventlist. This may result from a missing flight plan (e.g., the flight originated from a UZ message). It could also occur because the flight plan and position update information are coming from different sources and asynchronously. In most cases, the reported flight position occurs between the first and last events.

There is a chance that the TO flight position is farther than a preset threshold limit from the route. If this occurs when the flight is over the ocean, where the distance between adjacent events is measured in thousands of miles, then the TO information is still usable for correction.

When *TOMprocessingforfdb* determines that the last actual event is impossible to identify, the eventlist update is bypassed and the FDB entry is replaced with the updated record.

If *TOMprocessingforfdb* positively identifies the last actual event, it calculates the distance between the reported flight position and the known route. If the flight position is too far away from the route, *TOMprocessingforfdb* calls *Updateeventlistforfarawayto* to update the eventlist; if the flight position is close enough to the route, it calls *Updateeventlistfortom* to update the eventlist. If successful, *TOMprocessingforfdb* sends update transactions to the TDB and FTM and replaces the FDB entry with the updated flight record. If the eventlist update is unsuccessful, *TOMprocessingforfdb* returns FALSE.

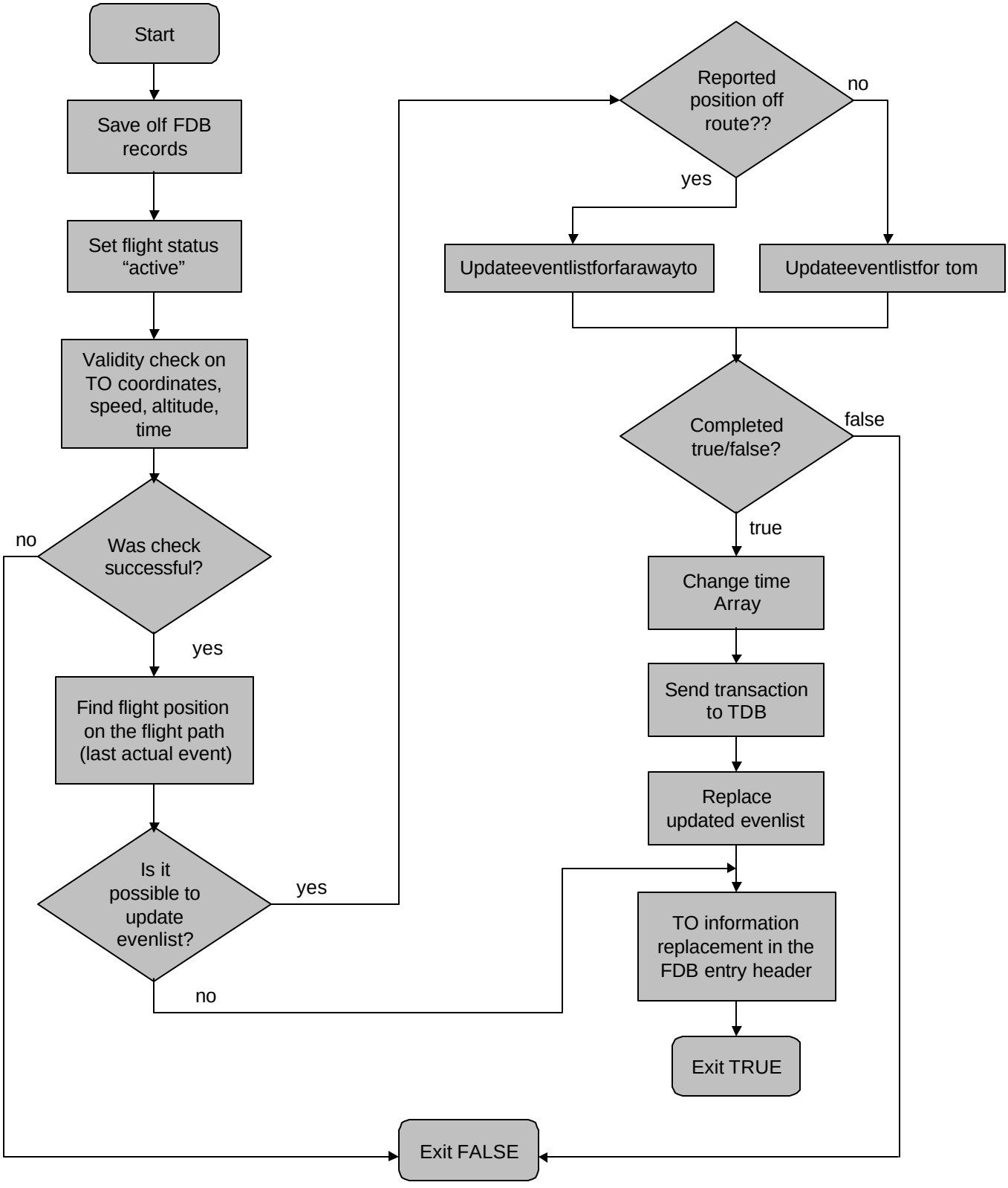

Figure 25-50. Sequential Logic for the TOMprocessingforfdb Module

25.14.4.5.13 The Updateeventlistfortom Module

The *Updateeventlistfortom* module updates an eventlist (velocity, altitude, time) using information provided by a TO message. The sequential logic for the *Updateeventlistfortom* function is shown in Figure 25-51 and described below.

These terms are used in the following description:

- **last actual event** (also called "prior event") –– This is the event in the event list most recently passed by this flight. It is determined based on the flight's current position.
- **previous last actual event** –– This is the most recent event in the event list for which a TZ time (not a modeled or interpolated time) is recorded.
- **correction event** –– This is the event in the event list that the flight is expected to reach next, normally the event after the last actual event.

The eventlist is updated for all events beginning with the event after the previous last actual event and up to the last event in the list. The first step is to find a correction event, i.e., an event which is ahead of the last actual event in the remaining flight path and belongs to an enroute phase of the flight. It is assumed that the TO messages are received when the flight is already in the enroute (cruising altitude) phase.

When a correction event can not be located because the flight has already entered the descent phase, the attempt to make a time correction is canceled and this routine terminates. If a correction event is successfully determined, the event list will be remodeled. The next step is to identify the previous last actual event, by searching through the event list for the last event with a time-type of TZ-time. Its time, velocity, and altitude will remain unchanged.

The flight is remodeled starting with the event following the previous last actual event. All enroute-phase events' speeds and altitudes are set to the current reported speed and altitude, and their estimated times are remodeled based on the new speed. The speeds of the approach/arrival/landing phase events are not changed, but their time estimates are re-calculated. The time-type of all of these recalculated times, from the event after the previous last actual event to the final event, are set to **modeled** in the event list.

The spherical distance from the reported position to the correction event is determined, and the new time for this event is calculated based on the reported speed. The previously modeled time is subtracted from this new time estimate, establishing a delta time. This delta will be added to every event over the rest of the eventlist beginning from the event after previous last actual event to the last event in the list. The time-types of events after the previous last actual event up?the event before the last actual event are set to **interpolated**; the time-type of the last actual event is set to TZ time; all subsequent time-types remain **modeled**. (Only **modeled** times will be subject to future recalculation.)

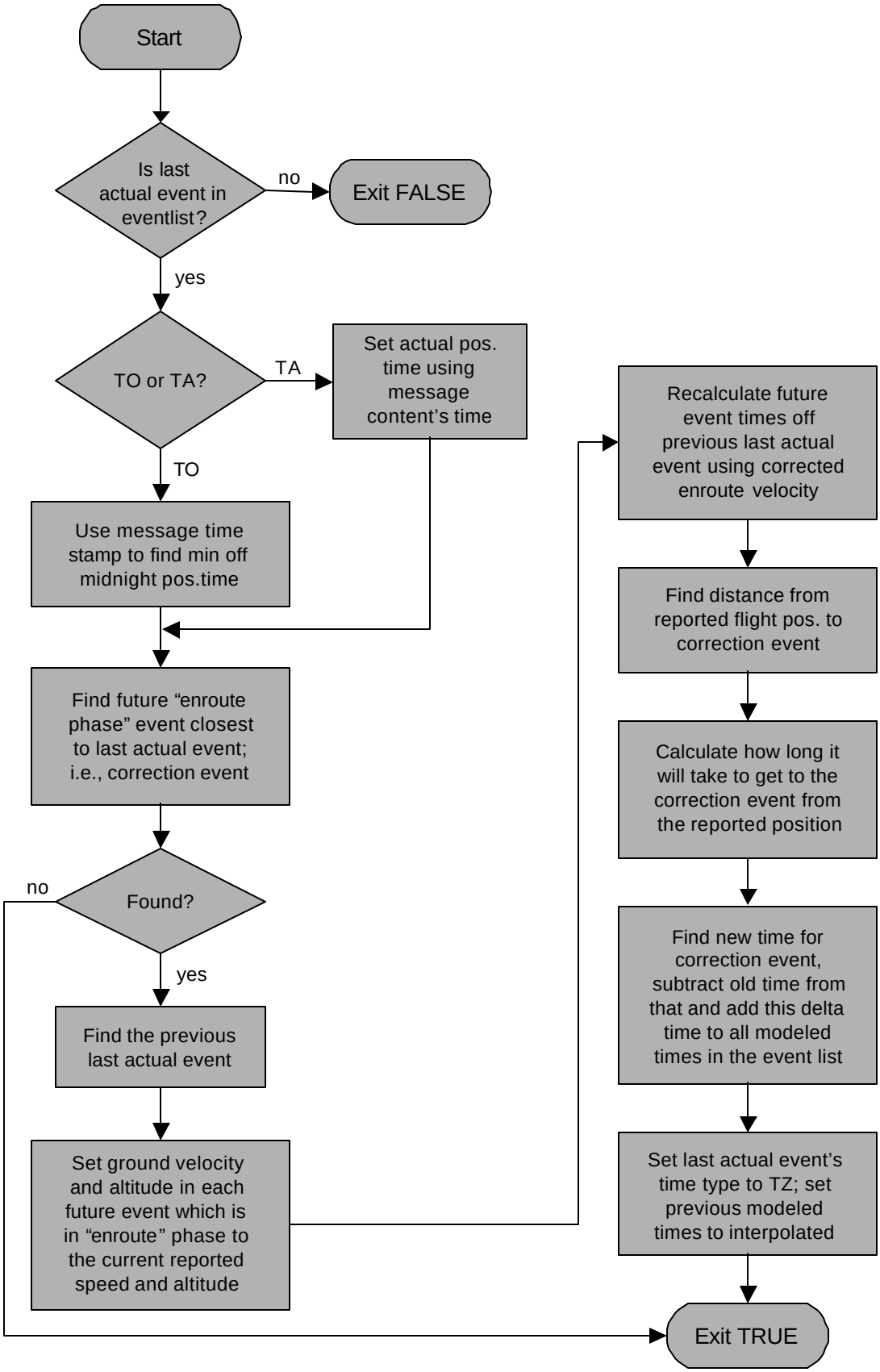

Figure 25-51. Sequential Logic for the Updateeventlistfortom Module

25.14.4.5.14 The Updateeventlistforfarawayto Module

The *Updateeventlistforfarawayto* module updates an eventlist (velocity, altitude, and time) using information provided by a TO message. The following describes the sequential logic for the *Updateeventlistforfarawayto* function and which is illustrated in Figure 25-52.

This routine is similar to *Updateeventlistfortom* (Section 25.14.4.5.13 defines some of the terms used in this section). It is called when the flight is very far off its route. When this is the case, trying to find the last actual event and the next expected event on the route's event list is not particularly meaningful, but it is still desirable to have an updated ETA. An assumption is made that the flight will attempt to get back onto its scheduled route by the time its descent phase begins, so the correction event is always assumed to be the last event of the enroute phase.

When a correction event cannot be located because the flight has already entered the descent phase, the attempt to make a time correction is canceled and this routine terminates. If a correction event is successfully determined, the event list will be remodeled. The next step is to identify the previous last actual event, by searching through the event list for the last event with a time-type of TZ time. Its time, velocity, and altitude will remain unchanged.

The flight is remodeled starting with the event following the previous last actual event. All enroute-phase events' speeds and altitudes are set to the current reported speed and altitude, and their estimated times are remodeled based on the new speed. The speeds of the approach/ arrival/landing phase events are not changed, but their time estimates are re-calculated. The time-type of all of these recalculated times, from the event after the previous last actual event to the final event, are set to “modeled” in the event list.

The spherical distance from the reported position to the correction event is determined, and the new time for this event is calculated based on the reported speed. The previously modeled time is subtracted from this new time estimate, establishing a delta time. This delta will be added to every event over the rest of the event list beginning from the event after previous last actual event to the last event in the list. Unlike time-types in *Updateeventlistfortom*, the time-types of all events being remodeled will remain “modeled”. They will eventually be re-calculated and set to “interpolated” when a TZ or TO is received from a position that is on or close to the scheduled route.

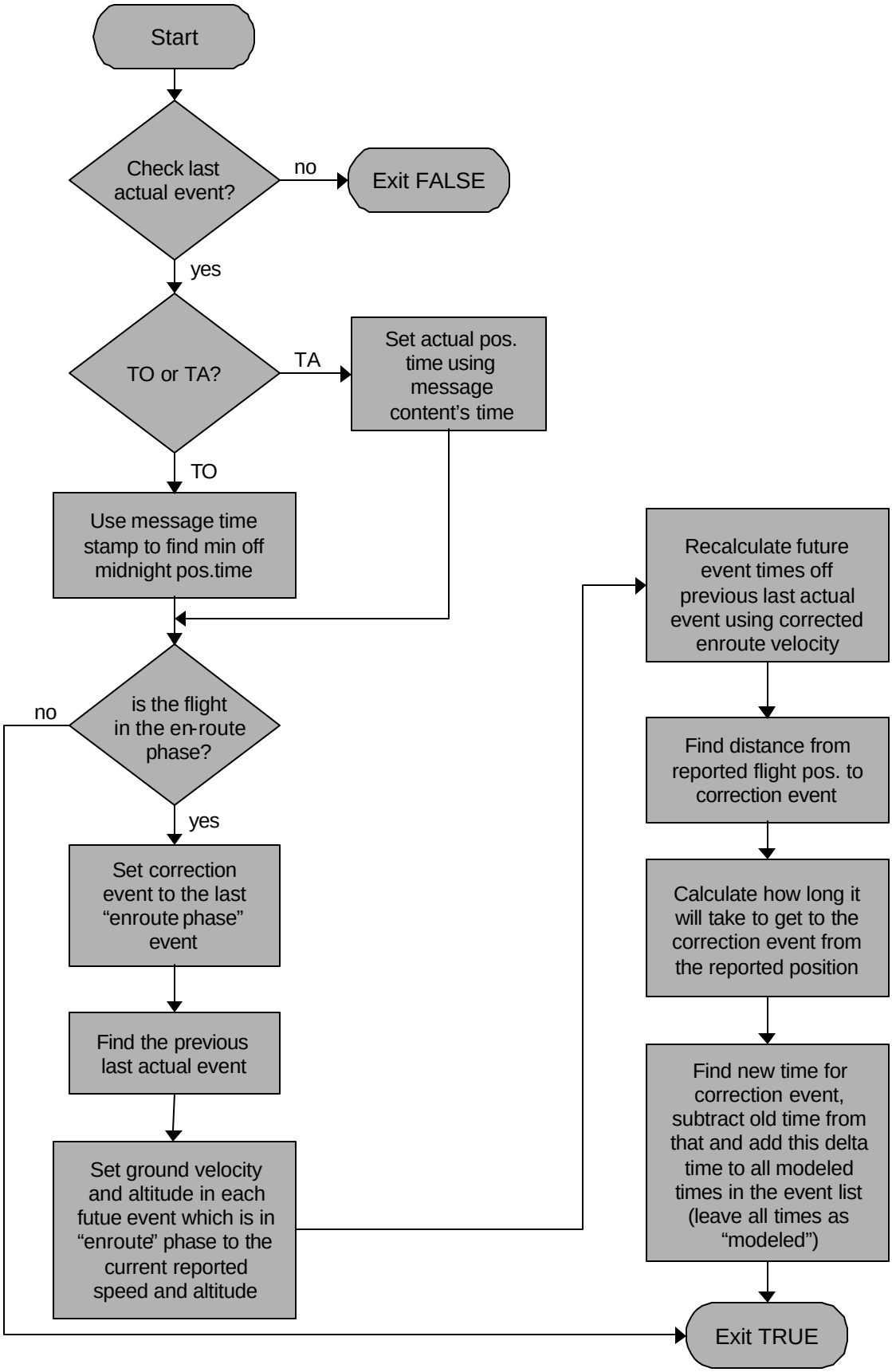

Figure 25-52. Sequential Logic for the Updateeventlistforfarawayto Module

25.14.4.5.15 The Do_AF Module

Do_AF processes two types of messages: AF (flight plan amendment) and FA (feedback). Since FA messages are derived from a combination of data already in the FDB and information contained in an AF message, concurrent processing makes message matching more consistent and also eliminates the need for redundant code. The sequential logic diagram for AF/FA messages is divided roughly in half. Figure 25-53 shows the processing for proposed messages and for active messages. In either case, the first step is to attempt a match between the flight ID in the message and at least one flight ID already in the database. If flight ID matching fails, the message is discarded. If at least one flight ID match is obtained, *Do_AF* then examines the message to determine what types of changes to the flight plan are being made (e.g., departure time, route, airspeed, altitude, etc.). During this examination, *Do_AF* sets internal flags that will be used later to decide how the message information will be processed.

For a proposed message, message matching begins by comparing the computer ID found in the message with all entries whose flight ID matches the message. If the computer ID match fails, *Do_AF* then checks for a match with **controlled**, **scheduled**, and **active** flights. Active message matching begins by comparing the message to **active** flights, followed by **controlled**, **filed**, and **scheduled** flights. If *Do_AF* finds that an active FA message matches a **controlled**, **filed**, or **scheduled** flight, it deactivates any currently active flights with the same flight ID as the message.

If the message is an AF that requests changes to the flight's route, altitude, or aircraft type, *Do_AF* places information from the matching entry's flight record into the message and enqueues a feedback message for the *Feedback Relay*. The *Parser* will reprocess the FA message using the additional information, and return the FA message to *Process Flight Messages*, which will perform the required updates to the database.

AF messages that do not require feedback message generation and all FA messages are processed by the *Replace AF/FA Information* module. Figure 25-54 shows that AF messages use a different set of processing steps than FA messages. Note that *Replace AF/FA Information* discards AF and FA messages for an entry that does not have an event list.

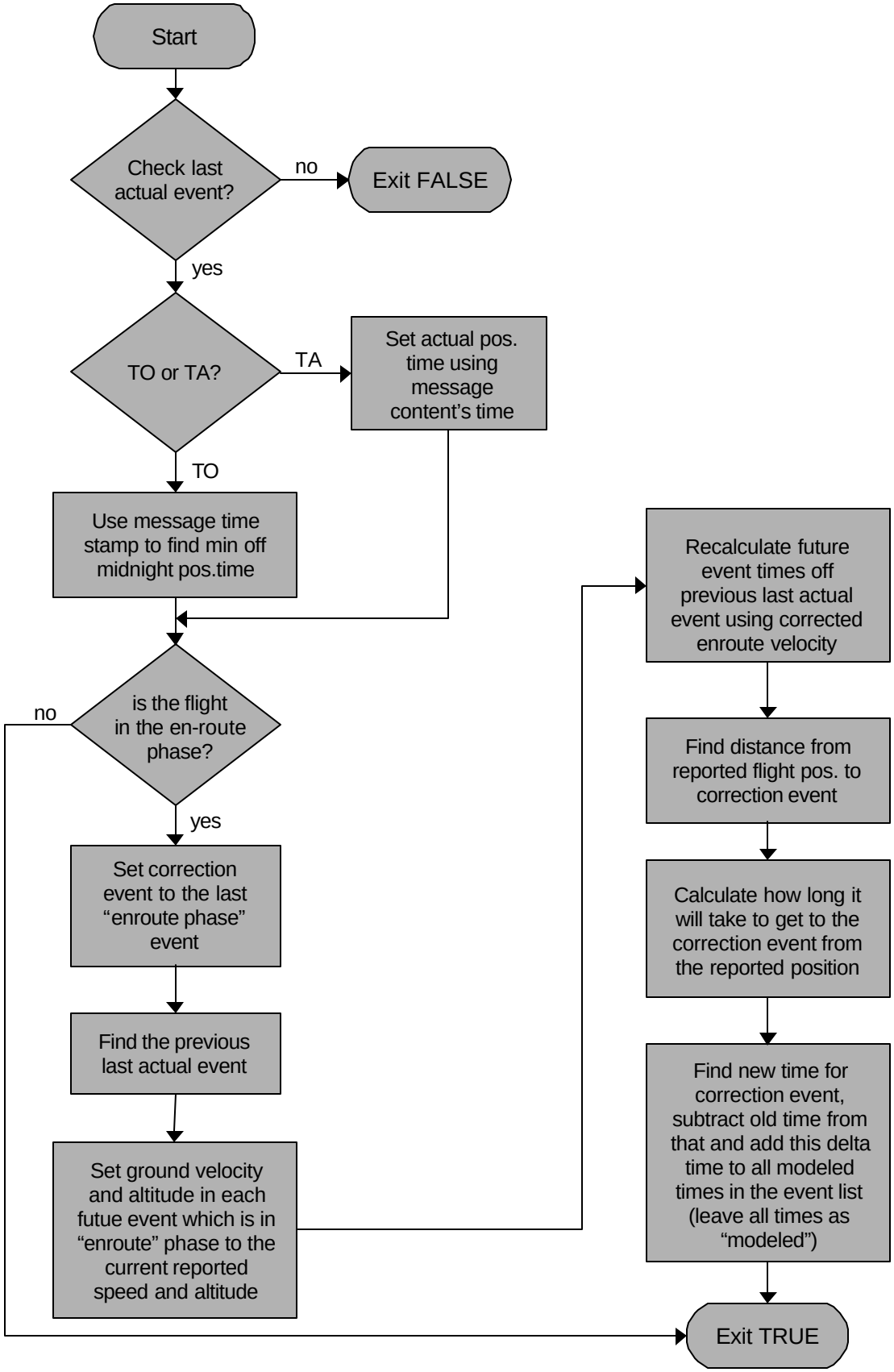

Figure 25-53. Sequential Logic for the Do_AF Module

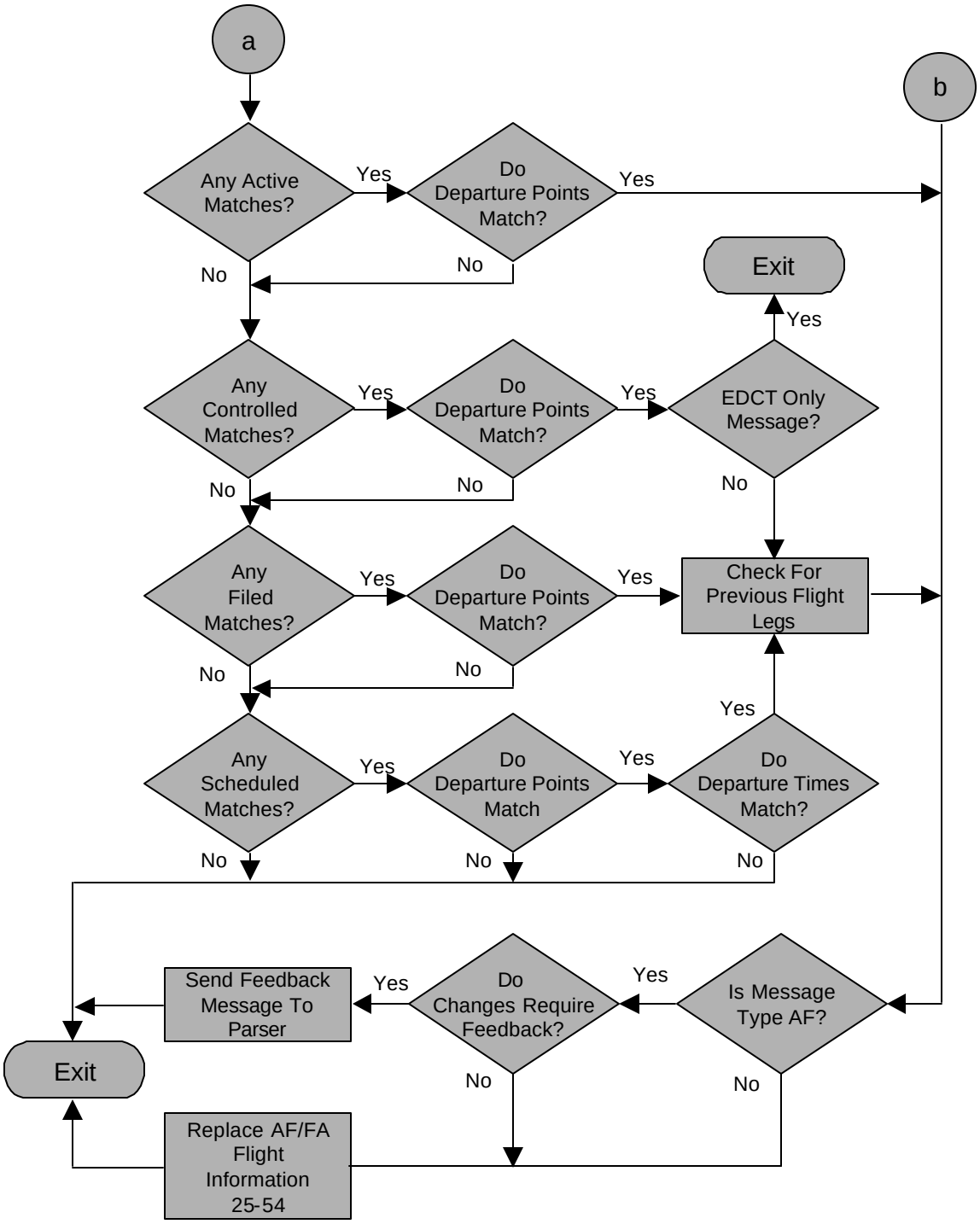

Figure 25-53. Sequential Logic for the Do_AF Module (continued)

When *Replace AF/FA Information* processes an AF message, it retrieves the matching entry's event list from the **evdb** map file and performs requested updates to the flight's departure time or cruising speed. To process an FA message, *Replace AF/FA Information* checks to see if the flight is **active** or if the field 10 contained a tailored route. If either condition is satisfied, *Replace AF/FA Information* merges the entry's event list with the event list in the message. Next, it makes any required change to the departure time.

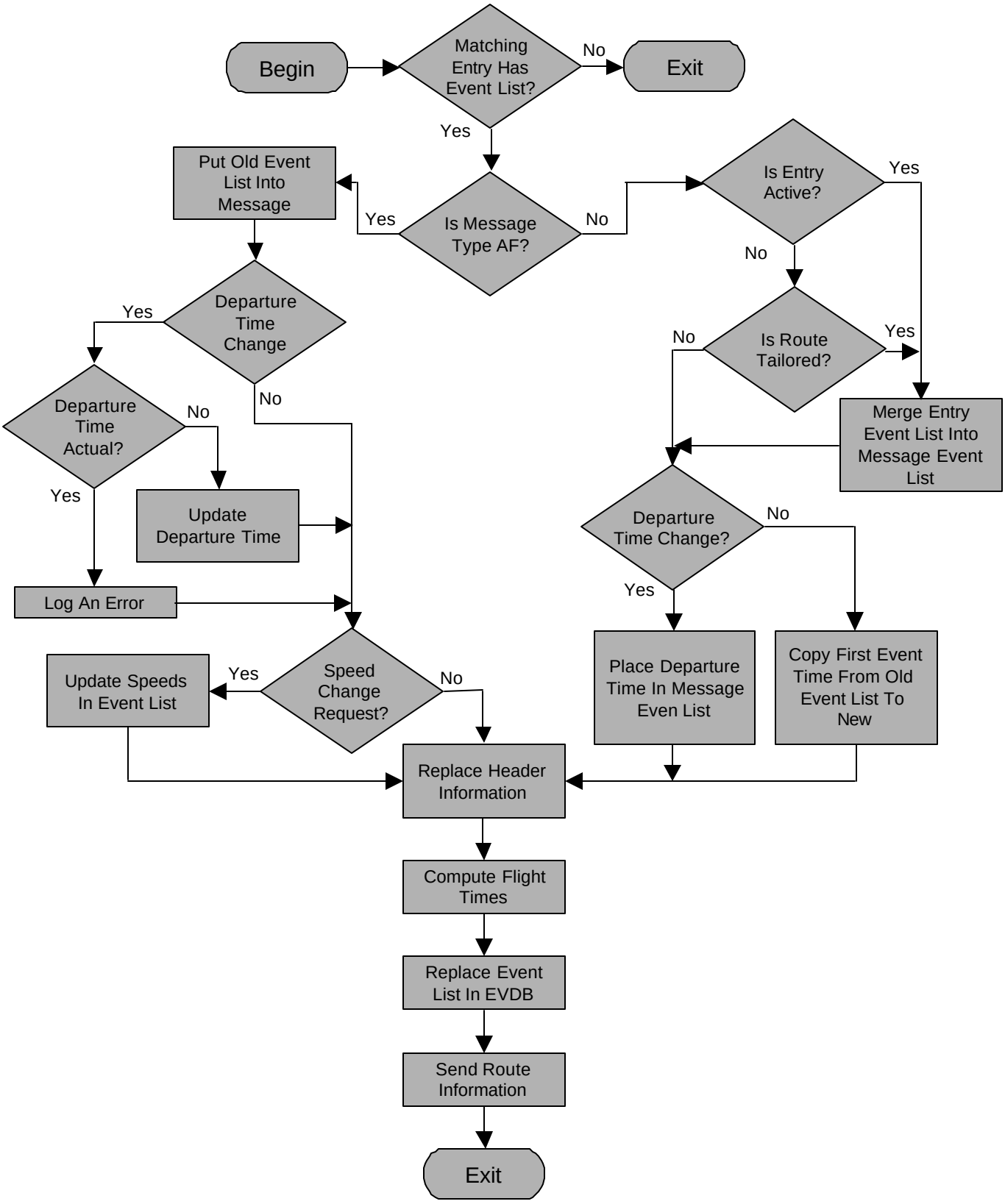

Figure 25-54. Sequential Logic for the Replace AF/FA Information Routine

At this point (for either message type), *Replace AF/FA Information* updates the information in the flight record header and computes the flight times. Flight time computation may take one of two different paths:

(1) For AF messages which requested a departure-time change but no speed change, a delta time value is applied to each of the current event list values.

(2) For FA messages and all AF messages not covered by (1), the event list times are recomputed by the *ModelFlight* module. Times are computed from the point of any event list merge to the end of the event list.

After computing the flight times, *Replace AF/FA Information* replaces the new event list. The update databases flag is set to true, and the TDB update type is set to replace.

25.14.4.5.16 The Do_AZ Module

Arrival (AZ) messages are processed by the *Do_AZ* module according to the sequential logic diagram of Figure 25-55. If the message flight ID does not match any of the flights currently in the database, *Do_AZ* adds a flight record and a single-event event list for the flight.

For messages which do match a flight ID currently in the database, *Do_AZ* checks all possible flight status values in the following order: **active**, **completed**, **controlled**, **filed**, **scheduled,** and **completed**.

The reason for the seemingly redundant check on completed flights is to allow *Do_AZ* to process arrival messages for flights that were internally deactivated by the *check_previous_flightlegs* routine. If the first check for **completed** flights fails, *Do_AZ* decides if the message matches other flight status values before proceeding to label the AZ message as redundant. If a redundant AZ message is detected, *Do_AZ* logs an error and discards the message.

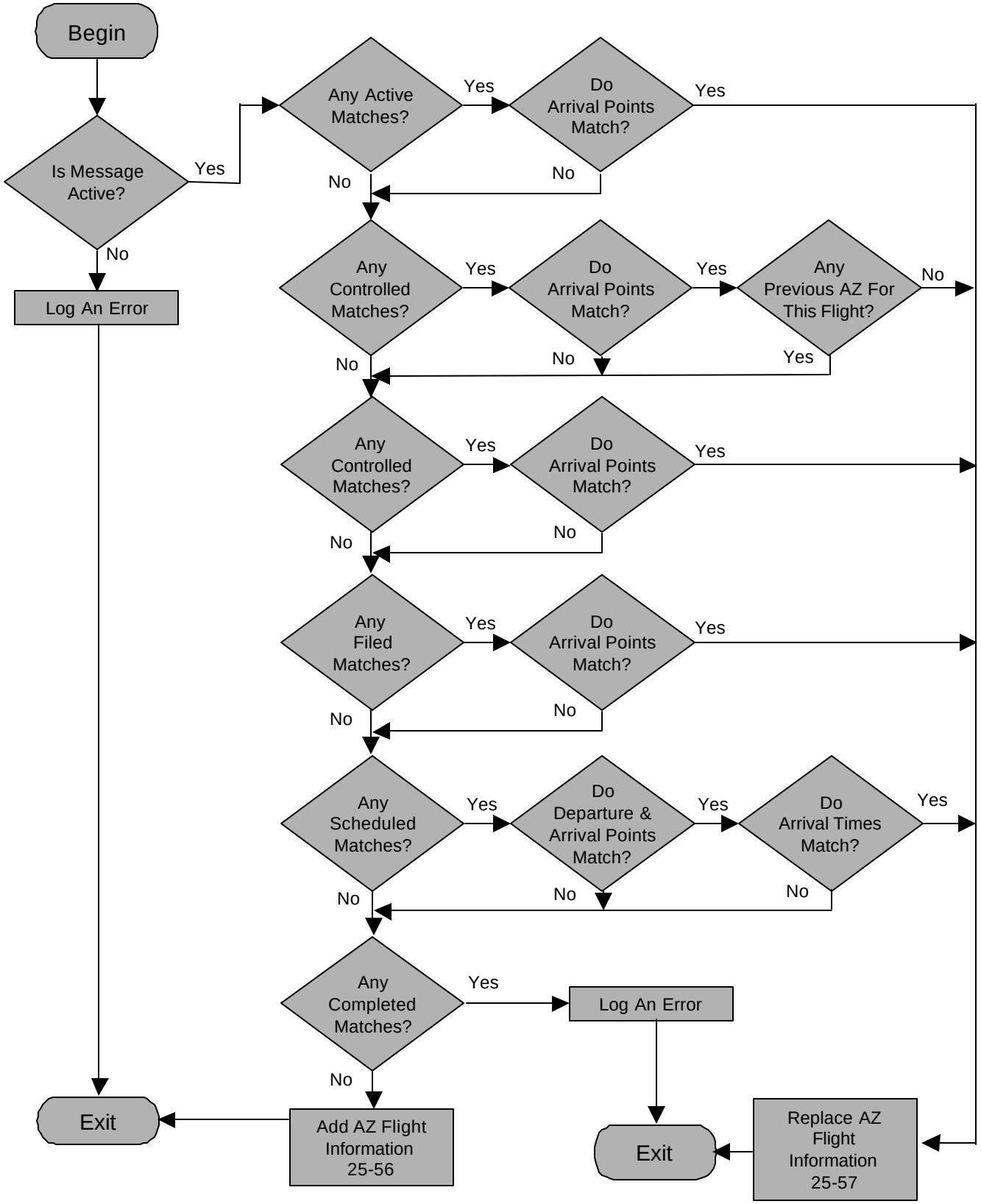

Figure 25-55. Sequential Logic for the Do_AZ Module

In order to add an AZ message to the database (see Figure 25-56), *Do_AZ* allocates and loads a new flight record header. It then creates a single-event event list for the flight. The update databases flag is set to true and the TDB update type is set to add.

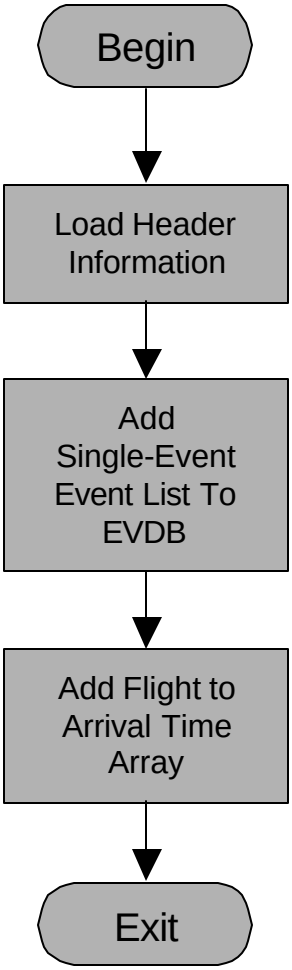

Figure 25-56. Sequential Logic for the Add AZ Information Routine

Do_AZ performs the logical steps depicted in Figure 25-57 to update flight information from an arrival message. If matching flight's event list already has an arrival event, *Do_AZ* merges the arrival event defined by the message with the arrival event already in the database. If no arrival event exists for the flight, *Do_AZ* appends the coordination fix event from the message to the end of the event list.

After the coordination fix has been merged or added, *Do_AZ* performs an interpolation of event times, which are previous to the arrival event but which are after any events in the event list that are already marked as proposed, actual, or interpolated. The update databases flag is set to true, and the TDB update type is set to replace.

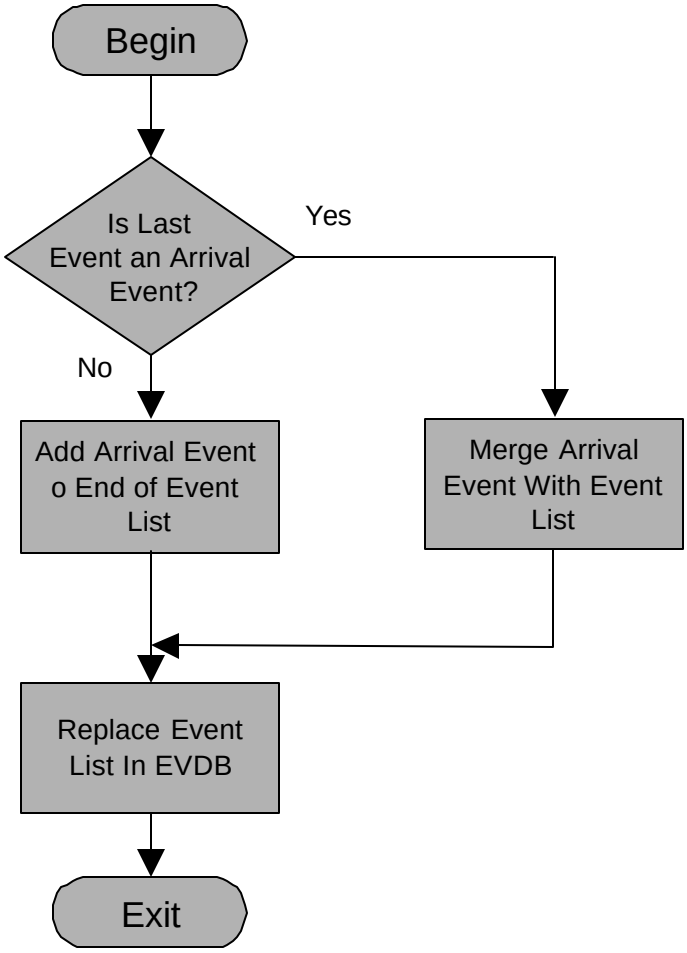

Figure 25-57. Sequential Logic for the Replace AZ Information Routine

25.14.4.5.17 The Do_RZ Module

Do_RZ processes RZ (flight cancellation) messages according to the logical sequence outlined in Figure 25-58. Since RZ messages are designed to cancel flights currently in the database, messages that do not match a flight ID currently in the database are discarded.

Proposed RZ messages are matched to filed**,** controlled, and scheduled flights, in that order. Active RZ messages may only match active flights. If a match is found, the flight status is set to cancelled, the update databases flag is set to true, and the TDB update type is set to delete.

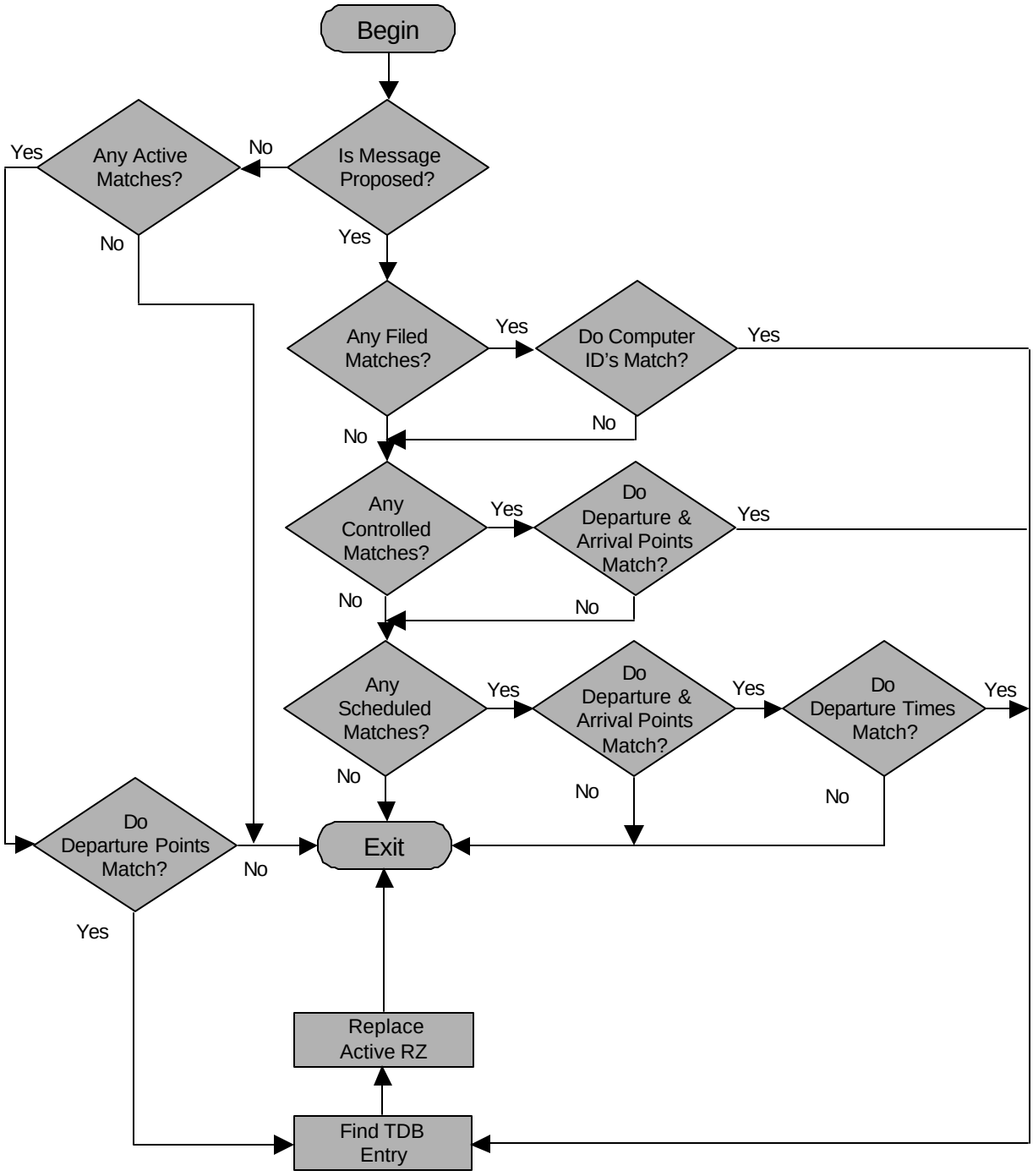

Figure 25-58. Sequential Logic for the Do_RZ Module

25.14.4.5.18 The Do_RS Module

Do_RS processes RS (schedule cancellation) messages according to the logical flow pictured in Figure 25-59. If *Do_RS* does not find a flight ID match in the database, the message is discarded. RS messages are, by design, aimed at canceling only those flights whose current status is scheduled. *Do_RS* generates an error message, if an RS message matches either a filed or active flight. If a scheduled flight match is found, the flight status is set to cancelled, the update databases flag is set to true, and the TDB update type is set to delete.

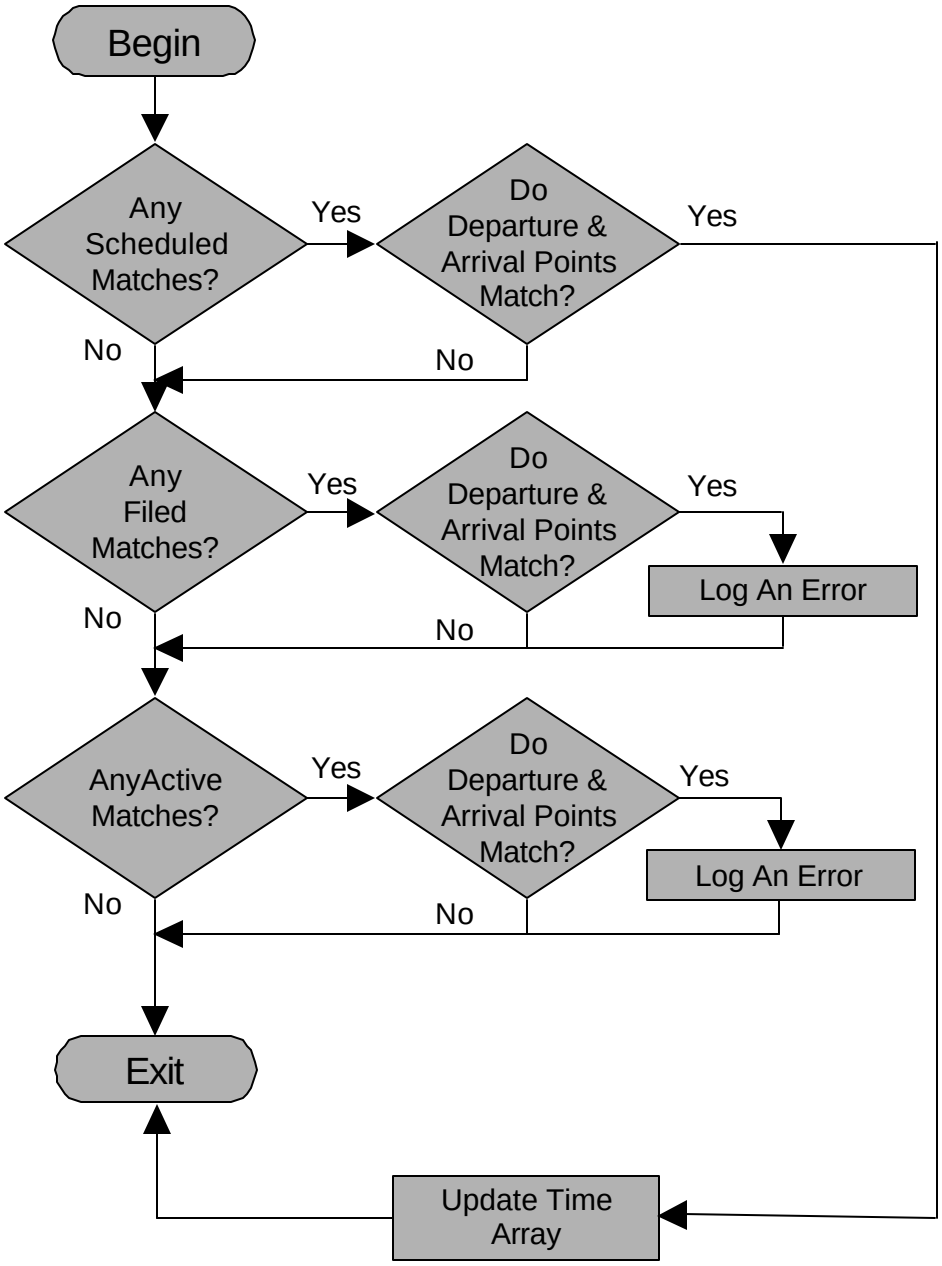

Figure 25-59. Sequential Logic for the Do_RS Module

25.15 Flight Database Processor Source Code Organization

The Flight Database Processor source code resides in C files under configuration management using ClearCase

25.16 Flight Database Processor Data Structures

This section describes data structures that are used by the *FDP*. Included are descriptions of structures internally used by the *FDP* and those used for communicating between this and other processes.

25.16.1 The flight_db_type Data Structure

The **flight_db_type** data structure is used as the format of each record in the FDB. All information about a particular flight, except for the flight's event list, is contained in this record. The **flight_db_type** is used exclusively by the *FDP* for updating the FDB. A more complete description of the data structure appears in Table 25-10 .

25.16.2 The event_block_type Data Structure

The **event_block_type** data structure is used as the format of each record in the event-list, mapped file part of the *FDB*. These records are used for storing the event lists of flights. An event list of a particular flight is stored in one to five records, depending on the list's length. The **event_block_type** data structure is a fixed length array of event record structures. The event record structure, referred to as the **erect** structure, is used by both the *Parser* and the *FDP*. The **erect** structure is shown in Table 25-11.

25.16.3 The fdb_ftm_interface_t Data Structure

The **fdb_ftm_interface_t** data structure is used by the *FDP* to communicate route information about a flight to the *Flight Table Manager*. This route information includes fixes, waypoints, and sectors through which the flight travels. A more detailed description of the data structure is shown in Table .

25.16.4 The fdb_tdb_interface_t Data Structure

The **fdb_tdb_interface_t** data structure is used by the *FDP* to communicate transactions to the *Traffic Demands Database Processor*. These transactions direct the *Traffic Demands Database Processor* to add, replace, or delete information about a flight's events. The transaction data structure is a variable length array of characters. Data items of various types (integers, characters, Pascal records) are placed into the array. The position of the item in the array helps determine what the data represents. The contents of the first field (the transaction letter) also determines what data appears in what position in the array. The fields which make up the transaction are shown in Table 25-12. Tables 25-13 and 25-14 describe the data structures used for the short event list.

Table 25-10. Flight_db_type Data Structure

Flight_db_type			
Library Name: gtp_openlib		**Element Name:** store_flushed_flts.h	
Purpose: to hold information about a particular flight			
Data Item	**Definition**	**Unit/Format**	**Var. Type**
deleted	Is this slot available for reuse?	--	boolean
tdbentry	Does this flight exist in the TDB?	--	boolean
tdb_num_events	Number of events in the TDB for this flight	_	short
id_of_flight	Flight identification	1 or 3 letters followed by numbers.	string7
comp_id	Computer identification for a flight	3 numbers	string3
ftm_flight_num	Date/time stamp for FTM and TDB transactions	Packed julian date and time in minutes	short
ftm_version_num	Not currently used	-	short
actype	NAS abbreviation for aircraft type	Combination of letters and numbers	string4
user_category	Category of user	An enumerated type: commercial, military, etc	usercat_t
flightreg_class	Flight regulations observed by the flight	An enumerated type: VFR, IFR_CAT1, etc.	FLIGHT_REGS_T
ac_general_class	General type of aircraft	An enumerated type: land, helicopter, etc.	actype_t
ac_cat_class	Specific type of aircraft	An enumerated type: civilian, jet, single piston_prop, etc	ac_cat_t
ac_weight_class	Aircraft weight class	An enumerated type: small, large, etc.	ac_weight_t
geographical _flags	Geographical filter flags	future use	short
fstatus	Status of the flight	An enumerated type: scheduled, filed, etc.	status_of_flight
last_msg_time	Time of last NAS message receipt.	Year, month, day, hour, and minute	CALCLOCK

Table 25-10. Flight_db_type Data Structure (continued)

Flight_db_type (continued)			
Data Item	**Definition**	**Unit/Format**	**Var. Type**
message_history	History of flight's NAS messages	Packed format containing counts for each message type.	short
fdb_indicator	16-bit flag field used internallt to the FDB.	Packed format, one bit for each of up to 16 flags	short
current_ctr	The ARTCC through which the flight is currently flying	One letter code representing the ARTCC.	char
route_text	Route of flight	ASCII field 10 string 256	
departure_sub	Subscript into departure time array	A number from 0 to 191	short
Arrival_sub	Subscript into appropriate arrival time array	A number from 0 to 191	short
profile_info	Record containing values for flight modeling routinea	Indices and distances to define aircraft dynamics	profile_rec_t
tz_indicator	Flags for TZ processing	16 one-bit flags, keep track of past TZ processing	short
tz-message_history	Number of TZ messages received	A number	short
to_message_history	Number of TA messages received	A number	short
ta_message_history	Number of TA messages received	A number	short
tz_delay_indicator	TZ delay information	16 bits	short
tz_time	Time when last TZ was received	Year, month, day, hour, and minute.	CALCLOCK
ta_time	Time when last TA was received.	Year, month, day, hour, and minute	CALCLOCK
oceanic_dist	Supports TO processing	Distance value	short
connection_event	Supports TO processing	Event number	short
last_act_lat	Last actual latitude	Radians times 10000	short
last_act_lon	Last actual longitude	Radians times 10000	short
lat_pos_predicted	Predicted latitude	Radians times 10000	short

Table 25-10. Flight_db_type Data Structure (continued)

Flight_db_type (continued)			
Data Item	**Definition**	**Unit/Format**	**Var. Type**
last_actual_event	Event number for plane's last reported position	Number between 0 and 250	short
last_actual_event	Event number for plane's last reported position	Number between 0 and 250	short
lon_pos_predicted	Predicted longitude	Radians times 10000	short
next_goal_event	Event number for plane's last reported position	Number between) and 250	short
fl_measured_heading	Flight heading measured between TZ messages.	Nadian times 10000	short
total_distance	Total distance covered by the event list.	Nautical miles	short
total_dist_strine	Total straight line distance flown.	Nautical miles	short
total_dist_passed	Total distance flown so far	Nautical miles	short
smooth_deviation	Smoothed deviation from route	Nautical miles	short
smooth_dev_slope	Smoothed slope of deviation value	Nautical miles per minute.	short
time_smooth_made	Time of last reevaluation in TZ processing	Year, month, day, hour, minute	CALCLOCK
proposed_speed	Requested air speed	Nautical miles per hour	short
reported_speed	Last reported ground speed	Nautical miles per hour	short
proposed_alt1	Requested altitude	Hundreds of feet	short
reported_alt1	Last reported altitude	Hundreds of feet	short
reported_altitude_type	Type of altitude	Transitional or mode C?	char
departure_ap	Airport of flight's origin	Combination of 3 or 4 letters and numbers	string4
arrival_ap	Destination airport of flight	Combination of 3 or 4 letters and numbers	string4
ground_time	Predicated number of minutes for flight to taxi	Minutes	short
dep_pushback	Predicted value for departure queue delay	Minutes	short

Table 25-10. Flight_db_type Data Structure (continued)

Flight_db_type (continued)			
Data Item	**Definition**	**Unit/Format**	**Var. Type**
departure_date	Julian date for flight's NAS messages	Number of days since January 1, 1980.	unsigned short
proposed_dep_time	Flight's proposed departure (from FZ)	Minutes from midnight	short
actual_dep_time	Flight's actual departure time (from DZ)	Minutes from midnight	short
sched_dep_time	Flight's scheduled departure time (from FS)	Minutes from midnight	short
control_dep_time	Flight's controlled departure time (from EDCT)	Minutes from midnight	short
first_event_time	Time from first event in the event list	Minutes from midnight	short
orig_dep_time	Latest proposed or scheduled dep time before control	Minutes from midnight	short
proposed_arr_time	Flight proposed arrival time (from FZ)	Minute from midnight	short
init_arr_time	Initial predictton for flight's arrival time	Minutes from midnight	short
curr_arr_time	Current prediction for flight's arrival time	Minutes from midnight	short
sched_arr_time	Flight's scheduled time of arrival (from FS)	Minutes from midnight	short
orig_arr_time	Latest proposed or scheduled arr time before control	Minutes from midnight	short
timestamp_offset	Time between time in DZ message and time stamp	Seconds	short
ground_time_method	Mode used in ground time determination	a=aircraft, c=category t=controlled, d=default	char
dept_center	Departure center code	Character or symbol	char
arr_center	Arrival Center code	Character or symbol	char
controllable	Indicated flight's ability to accept control programs	_	Boolean
strategy_flags	Flags for strategy processing	Not currently used	short
cta_list	Events describing controlled times of arrival	two events	array of short_erect

Table 25-10. Flight_db_type Data Structure (continued)

Flight_db_type (continued)			
Data Item	**Definition**	**Unit/Format**	**Var. Type**
arrival_fix_event	Event to describe the flight's arrival fix	Location and time of arrival fix	short_erect
ac_rmk_bitflags	Flags for field 11 key-word	16-bit flags	short
numb_of_events	Size of flight's event list	A number from 0 to MAXEVENTS	short
elist_offset	Offset into EVDB for this flight	Number of bytes from beginning of EVDB map file	INT32
last_pos_message_to	Indicated whether last pos message was a TO	_	Boolean
proposed_altitude_type	Type of altitude: Block, OTP, AVR, etc.	a character	char
proposed_alt2	Request altitude 2	Hundreds of feet	short
reported_alt2	Last reported altitude 2	Hundreds of feet	short
num_aircraft	Number of aircraft in formation	From field 3	char
ac_eqp_prefix	aircraft equipment pre-fix	From field 3	char
ac_eqp_suffix	aircraft equipment suffix	From field 3	char
az_source	Source indicator of arrival message	"A" = ARTS "E" = Host	char
new_fid_flag	Flag indicating a flight ID change	flag	Boolean
tz_proc_flag	Indicator of TZ proces-sing method		char

Table 25-11. erect Data Structure

erect (event record type)

Library Name: ttm_openlib | **Element Name:** event.h

Purpose: To contain information about an event. The contents and type of each data item is shown here.

Field Name: time_index **Field Type:** INT32

Data Item	Definition	Unit/Range	Which Bits?
event kind	Is this event an arrival or departure?	0 - 2	31 - 30
phase	In which phase of the flight does the event occur?	Enumerated type from TAKEOFF to LANDING (0-6)	29 - 27
time	At what time does this event occur?	Minutes from midnight	26 - 15
TDB time type	Is this an actual or predicted event?	1 means actual, 0 means predicted	14
time type	Desc. the type of the event time (actual, predicted).	Constants defining time types range from 0 - 7	13 - 11
unused	--	--	10 - 0

Field Name: del_alt-vel **Field Type:** INT32

Data Item	Definition	Unit/Range	Which Bits?
delay	Any filed airborne delay.	Minutes	31 – 22
altitude	The altitude of this flight at this event.	Flight level in hundreds of feet	21 – 12
velocity	The velocity of this flight at this event.	Nautical miles per minute	11-0

Field Name: distance **Field Type:** short

Data Item	Definition	Unit/Range	Which Bits?
waypoint flag	Is this position of this event a waypoint of the flight?	1 means yes, 0 no	15
unused	--	--	14
distance	The distance this flight has flown from the last event.	Nautical miles	13 - 0

Table 25-11. erect Data Structures (continued)

erect (continued)

Field Name: heading_type **Field Type:** short

Data Item	Definition	Unit/Range	Which Bits?
monitor flag	Is the element of this event monitored	1 means yes; 0 means no	15
heading	Heading of this flight at this event	0 – 359 degrees	14 – 6
element type	At what type of element does this event occur?	These types currently range from 0 to 18	5 - 0

Field Name: element index **Field Type:** short

Data Item	Definition	Unit/Range	Which Bits?
element_index	Element's index in the grid database	0 – 65535	15 - 0

Field Name: latitude **Field Type:** short

Data Item	Definition	Unit/Range	Which Bits?
latitude	Latitude of the position of this event	radians time 1000	15 - 0

Field Name: longitude **Field Type:** short

Data Item	Definition	Unit/Range	Which Bits?
longitude	Longitude of the position of the event	radians times 1000	15 - 0

Table 25-12. Logical structure for the FDB to TDB Interface

FDB—TDB Communications Interface				
Library Name: traffic_openlib	**Purpose:** This variable structure describes the transaction format used to pass information to the TDB. The FDB sends flight update (add, replace,delete), time, and FA Flight Request transactions to the TDB.			
Element Name: fdbtdb.interface.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
transaction type	Letter identifying the transaction type.		*	char
For Time transactions:				
filler	Filler character to ensure that next field begins on even byte.			char
current time	Current Greenwich mean time.	System type		CALCLOCK
For Flight transactions:				
flight status char	Character of flag bits representing flight status information.	Flags bits. 5 are currently used. +		char
flight id	Set of characters indentifying this flight.	1 to 3 letters, then numbers.		7 characters
computer id	Internal computer ID of the flight.	3 numbers.		3 characters
flight number	A unique numeric flight identifier to be sent to the TDB.		0 – 65535	unassigned short
filler	Filler for alignment			short
For Deletes:				
old event list	List of events currently in the TDB that need to be deleted.	Variable length event list.	See following Table.	evenlist_t
For Adds, Replaces::				
tdb header record	Header record for future use.			tdb_headerinfo_rec_t
For Adds:				
new event list	List of flight events to be added to the TDB.	variable length events list.	See following Table.	eventlist_t
For Replaces:				
old event list	List of flight events to be deleted from the TDB.	variable length event_list.	See following Table.	eventlist_t
new even list	List of flight events for the same flight to be added to the TDB.	variable length event list.	See following Table.	eventlist_t
For FA Flight Request transactions:				
filler	Filler character to ensure that next field begins on even byte			char
nas_time	Element index for airport			short
start_time	Request start time			CALTIME
end_time	Request end time			CALTIME

+ bit flight status
- 0 = old event list active
- 1 = new event list active
- 2 = old event list scheduled
- 3 = new event list scheduled
- 4 = flight is non-commercial

* type
- 'A' = Add Flight
- 'D' = Delete Flight
- 'R' = Replace Flight
- 'T' = Time Transaction
- 'L' = FA List Request

Table 25-13. eventlist_t Data Structure

eventlist_t				
Library Name: traffic_openlib		**Purpose:**		
Element Name: short_event_list.h		To hold a single eventlist. This list supports FDB-TDB communications.		
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
jdate	Julian departure date of the flight	# of days since January 1, 1980	0 – 65535	unsigned short
length	The number of events in the following list		0 – 300	short
list	The list of events making up the flight's route	Array of Short erect records	0 – 300 elements	ev_tdb_array

Table 25-14. eventlist_t Data Structure

short_erect				
Library Name: tmm_openlib	**Purpose:** To contain information about an event. This structure is used for FDB-TDB communications. It is also a vehicle to pass arrival fix information from the Parser to the FDB.			
Element Name: short_event.h				
Data Item	**Definition**	**Unit/Format**	**Definition**	**Var. Type/Bits**
time_index	Packed structure that contains first part of event data.			INT32
For Airports:				
event kind	Item that indicated whether the event is arrival or departure	0 = departure 1 = arrival	0 – 1	31 -30
For Fixes:				
event kind	Item that indicates altitude of fix crossing	0 = low , 1= high 2 = superhigh	0 – 2	31 - 30
For Secotrs:				
event kind	Item that indicates whether this event is an entry or exit	0 = exit 1 = entry	0 – 1	31 - 30
phase	Flight phase at which this event occurs	Enumerated type	0 – 7	29 - 27
time	Time at which this event oc-curs	Minutes after midnight	0 – 2879	26 -15
TDB time type	Item that indicated whether this event is actual or predicted	0 = predicted 1 = actual	0 – 1	14
FDB time type	Item that indicates method usedto determine time	Integer constant	0 – 7	13 - 11
unused	These 11 bits are currently un-used			10 - 0
heading_type	Packed structure that contains the rest of the event data			INT32
monitor flag	Flag that tells whether this event is monitored for alerts	0 = no, 1 = yes	0 – 1	31
heading	Heading of this flight at this event	Degrees	0 – 359	30 - 22
element type	Type of element at which this event occurs	Integer constant	0 – 18	21 - 16
element index	Element's index in the grid da-tabase	Grid database in-dex	0 – 65535	15 - 0

25.16.5 FDB_FTM Data Structure

The *Fdb_ftm data structure* is composed of seven separate structures (see Tables 25–15 through 25–27). These structures are defined as either fixed or variable. Fixed structures are of a set length whereas variable structures are composed of one part of a set length and one part of changeable length. Of the seven data structures that follow, the *Block data structure* and the *Route data structure* are variable. All others are fixed.

An eighth structure, *Block_altitude data structure*, is defined in **ftm_tsc_interface** (see Section 18).

Table 25-15. TTM-FTM Block Transaction Data Structure

TTM-FTM block Transaction				
Library Name: fdb_openlib **Element Name:** fdbftm_pack.openlib		**Purpose:** Hold a complete block of data for transfer from the FDB to client FTMs		
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
flight_indx	Internal identifer.			INT32
tstamp	Time of message	Seconds after midnight	0-86399	INT32
fstat_spd_alt	Packed flight status, speed & altitude	See below		INT32
sizes	Packed number of waypoint, sector, fixes, airways ,& centers	See below		INT32
julian_date	Departure date	Number of days since 1/1/1980		short
std	Scheduled departure time	Minutes after midnight	0-1439	short
sta	Scheduled arrival time	Minutes after midnight	0-1439	short
ptd	Proposed departure time	Minutes after midnight	0-1439	short
pda	Proposed arrival time	Minutes after midnight	0-1439	short
td	Departure time	Minutes after miidnight	0-1439	short
ta	Arrival time	Minutes after midnight	0-1439	short
etd	Estimated departure time	Minutes after midnight	0-1439	short
eta	Estimated arrival time	Minutes after midnight	0-1439	short
ctd	Controlled departure time	Minutes after midnight	0-1439	short
cta	Controlled arrival time	Minutes after midnight	0-1439	short
otd	Original gate time of departure	Minutes after midnight	0-1439	short
ota	Original gate time of arrival	Minutes after midnight	0-1439	short

Table 25-15. TTM-FTM Block Transaction Data Structure (continued)

TTM-FTM block Transaction (continued)				
Library Name: fdb_openlib		**Purpose:** Hold a complete block of data for transfer from the FDB to client FTMs		
Element Name: fdbftm_pack.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
lat	Last reported latitude	Minutes of arc	-32768 –5400	short
lon	Last reported longitude	Minutes of arc	32768 –10800	short
nxt_lat	Latitude of the next event	Minutes of arc	-32768 –5400	short
nxt_lon	Longitude of the next event	Minutes of arc	-32768 –10800	short
seq_no	Sequence number for batch re-quests			short
ete	Estimated time enroute from current position	Minutes after midnight	0-1439	short
arr_fix_time	Estimated time at arrival fix	Minutes after midnight	0-1439	short
etime	Last message time	Minutes after midnight	0-1439	short
rte_size	Number of characters in route		0-1439	short
co_addr	FTM coprocess	See net$_ad_ dress		NET_ADDRESS_T
flight_id	aircraft identifie.r	1 to 3 letters and up to 4 numbers		string7
arr_fix	Arrival fix		0-1439	string6
dept_aprt	Airport of origin	3 to 5 letters & numbers		string5
arr_aprt	Destination airport	3 to 5 letters & numbers		string5
acft_type	Type of aircraft		unkn, land, heli, amph, sea	string4
dept_ctr	Departure center code	See ARTCC cen- ter codes	0-1439	char
arr_ctr	Arrival center codes	See ARTCC cen- ter codes		char
classes	Packed waypoint flag & physi-cal, user, and weight class	See below		char
altitude _type	Either trasitioning or cruising mode		C,T	char
num _type	Number of aircraft in formation	From field 3		char
ac_eqp_prefix	Aircraft equipment prefix	From field 3		char
ac_eqp_prefix	Aircraft equipment suffix	See below		char
az_source	Source indicator of arrival message	A = ARTS E = Host	A, E	char

Table 25-15. TTM-FTM Block Transaction Data Structure (continued)

TTM-FTM block Transaction (continued)				
Library Name: fdb_openlib **Element Name:** fdbftm_pack.h	**Purpose:** Hold a complete block of data for transfer from the FDB to client FTMs			
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
ac_rmk_bitflags	Flags for field 11 keywords	16 bitflags	0.1	short
geographical_flags	Flags for suture use	unused		short
pad	Space holder to keep structure an even size			char
rte_txt	String containing wypts,sctrs, arwys,fixes,ctr, and field 10	tot_route_t. See ftm tsc interface	up to 1466 char	char

Note: The above data structure is a variable data structure. The variable portion is **rte_txt**. Everything prior to **rte_txt** is fixed length.

Table 25-16. TTM-FTM Route Transaction Data Structure

TTM-FTM Route Transaction				
Library Name: fdb_openlib		**Purpose:** Hold route data for transfer from FDB to client FTMs		
Element Name: fdbftm_pack.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
flight_indx	Internal identifer.			INT32
tstamp	Time of message	Seconds after midnight	0-86399	INT32
fstat_spd_alt	Packed flight status, speed & altitude	See below		INT32
size	Packed number of waypoints, sector, fixes, airways & centers	See below		INT32
julian_date	Departure date	Number of days since 1/1/1980		short
etd	Estimated departure time	Minutes after midnight	0-1439	short
eta	Estimated arrival time	Minutes after midnight	0-1439	short
td	Departure time	Minutes after midnight	0-1439	short
ta	Arrival time	Minutes after midnight	0-1439	short
nxt_lat	Latitude of the next event	Minutes of arc	-32768 – 5400	short
nxt_lon	Longitude of the next event	Minutes of arc	-32768 – 10800	short
arr_fix_time	Estimated time at arrival fix	Minutes after midnight	0-1439	short
rte_size	Number of characters in route		0-1466	short
flight_id	Aircraft ASCII indentifier	1 or 3 letters and up to 4 numbers		string7
arr_fix	Arrival fix			string6
dept_aprt	Airport of origin	3 to 5 letters & numbers		string5
arr_aprt	Destination airport	3 to 5 letters & numbers		string5
acft_type	Type of aircraft	Land, copter, etc	unkn, land, heli, amph, sea	string4
srce	Source NAS message type	See below	F,U,A,S,Y char	

Table 25-16. TTM-FTM Route Transaction Data Structure (continued)

TTM-FTM Route Transaction (continued)

Library Name: fdb_openlib	**Purpose:** Hold route data for transfer from FDB to client FTMs
Element Name: fdbftm_pack.h	

Data Item	Definition	Unit/Format	Range	Var. Type/Bits
dept_arr_flgs	Packed time types of depar- ture and arrival	See below		char
dept_ctr	Departure center code	See ARTCC cen- ter codes		char
arr_ctr	Arrival center code	See ARTCC cen- ter codes		char
classes	Packed waypoint flag & physi- cal, user, and weight class	See below		char
altitude_type	Either transitioning or cruising mode			char
num_aircraft	Number of aircraft in formation	from field 3		char
ac_eqp_prefix	Aircraft equipment prefix	From field 3		char
ac_eqp_suffix	Aircraft equipment suffix	See below		char
ac_rmk_bittflags	Flags for field 11 keywords	16 bittflags	0.1	short
geographical_flags	Flags for future use	unused		short
pad	Space holder to keep structure and even size			char
rte_txt	String containing wypts, sctrs, arwys, fixes, ctrs, and field 10	tot_rounte_t See ftm tsc interface	up to 1466 char	char

Note: The above data structure is a variable data structure. The variable portion is **rte_txt**. Everything prior to **rte_txt** is fixed length.

The **srce** field can have one of the following values:

F=FZ

U=UZ

A=AF

S=FS

Y=FY

Table 25-17. TTM-FTM Recovery Transaction Data Structure

TTM-FTM Recovery Transaction				
Library Name: fdb_openlib **Element Name:** fdbftm_pack.h		**Purpose:** Hold recovery data for transfer from FDB to client FTMs		
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
flight_indx	Internal identifer.			INT32
tstamp	Time of message	Seconds after midnight	0-86399	INT32
fstat_spd_alt	Packed flight status, speed & altitude	See below		INT32
size	Packed number of waypoints, sector, fixes, airways & centers	See below		INT32
julian_date	Departure date	Number of days since 1/1/1980		short
lat	Last reported latitude	Minutes of arc	-32768 –5400	short
lon	Last reported longitude	Minutes of arc	-32768 –5400	short
nxt_lat	Latitude of the next event	Minutes of arc	-32768 –5400	short
nxt_lon	Longitude of the next event	Minutes of arc	-32768 –5400	short
etd	Estimated departure time	Minutes after midnight	0-1439	short
eta	Estimated arrival time	Minutes after midnight	0-1439	short
ctd	Controlled departure time	Minutes after midnight	0-1439	short
cta	Controlled arrival time	Minutes after midnight	0-1439	short
otd	Original gate time of depar-ture	Minutes after midnight	0-1439	short
ota	Original gate time of arrival	Minutes after midnight	0-1439	short
etime	Last message time	Minutes after midnight	0-1439	short
ete	Estimated time enroute from current position	Minutes after midnight	0-1439	short
arr_fix_time	Estimated time at arrival fix	Minutes after midnight	0-1439	short
flight_id	Aircraft ASCII indentifier	1 or 3 letters and up to 4 numbers		string7
arr_fix	Arrival fix			string6
dept_aprt	Airport of origin	3 to 5 letters & numbers		string5
arr_aprt	Destination airport	3 to 5 letters & numbers		string5
acft_type	Type of aircraft	Land, copter, etc	unkn, land, heli, amph, sea	string4

Table 25-17. TTM-FTM Recovery Transaction Data Structure (continued)

TTM-FTM Recovery Transaction (continued)				
Library Name: fdb_openlib		**Purpose:** Hold recovery data for transfer from FDB to client FTMs		
Element Name: fdbftm_pack.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
dept_ctr	Departure center code	See ARTCC center codes		char
arr_ctr	Arrival center code	See ARTCC center codes		char
classes	Packed waypoint flag & physi- cal, user, and weight class	See below		char
altitude_type	Either transitioning or cruising mode			char
num_aircraft	Number of aircraft in formation	from field 3		char
ac_eqp_prefix	Aircraft equipment prefix	From field 3		char
ac_eqp_suffix	Aircraft equipment suffix	See below		char
az_source	Source indicator of arrival mes- sage	A=ARTS E=Host	A<E	char
ac_rmk_bittflags	Flags for field 11 keywords	16 bittflags	0.1	short
geographical_flags	Flags for future use	unused		short
pad	Space holder to keep structure and even size			char

Table 25-18. TTM-FTM Cancel Transaction Data Structure

TTM-FTM Cancel Transaction				
Library Name: fdb_openlib		**Purpose:** Hold cancellation data for transfer from FDB to client FTMs		
Element Name: fdbftm_pack.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
flight_indx	Internal identifer.			INT32
tstamp	Time of message	Seconds after midnight	0-86399	INT32
julian_date	Departure date	Number of days since 1/1/1980		short
flight_id	Aircraft indetifer	1 or 3 letters and up to 4 numbers		string7
srce	Source NAS message type	See below	R,Z,C,X,H	char
geographical_flags	Flags for future use	unused		short

The **srce** field can have one of the following values:

R=RS

Z=RZ

C=RY

X=SI CANCEL

H=CONTROL CANCEL

Table 25-19. TTM-FTM Position Transaction Data Structure

TTM-FTM Position Transaction				
Library Name: fdb_openlib	**Purpose:** Hold position data for transfer from FDB to client FTMs			
Element Name: fdbftm_pack.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
flight_indx	Internal identifer.			INT32
tstamp	Time of message	Seconds after midnight	0-86399	INT32
spd_alt	Packed speed & altitude	See below		INT32
julian_date	Departure date	Number of days since 1/1/1980		short
lat	Last reported latitude	Minutes of arc	-32768 –5400	short
lon	Last reported longitude	Minutes of arc	-32768 –10800	short
nxt_lat	Latitude of the next event	Minutes of arc	-32768 –5400	short
nxt_lon	Longitude of the next event	Minutes of arc	-32768 –10800	short
lat_ii	Latitude from 2nd event in NAS message	Minutes of arc	-32768 –5400	short
lon_ii	Longitude from 2nd event in NAS message	Minutes of arc.	32768 –10800	short
lat_iii	Latitude from 3rd event in NAS message	Minutes of arc	-32768 –5400	short
lon_iii	Longitude from 3rd event in NAS message	Minutes of arc	-32768 –10800	short
eta	Estimated arrival time	Minutes after midnight	0-1439	short
etime	Last message time	Minutes after midnight	0-1439	short
etime_ii	Time of 2nd event in NAS message	Minutes after midnight	0-1439	short
etime_iii	Time of 3rd event in NAS message	Minutes after midnight	0-1439	short
flight_id	Aircraft ASCII indentifier	1 or 3 letters and up to 4 numbers		string7
dept_aprt	Airport of origin	3 to 5 letters & numbers		string5
arr_aprt	Destination airport	3 to 5 letters & numbers		string5
srce	Source NAS message type	o=TO, W=TA	O,W	char

Table 25-19. TTM-FTM Position Transaction Data Structure (continued)

TTM-FTM Position Transaction (continued)				
Library Name: fdb_openlib		**Purpose:** Hold position data for transfer from FDB to client FTMs		
Element Name: fdbftm_pack.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
wypt_flag	Ghosting flag	0 =Ghost to waypoint	0-1	char
altitude_type	Either transitioning or cruising mode		C,T	char
arr_fix_time	Estimated time at arrival fix	Minutes after midnight	0-1439	short
geographical_flag	Flags for future use	unused	unused	short

Table 25-20. TTM-FTM Time Transaction Data Structure

TTM-FTM Time Transaction				
Library Name: fdb_openlib		**Purpose:** Hold time data for transfer from FDB to client FTMs		
Element Name: fdbftm_pack.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
flight_indx	Internal identifer.			INT32
tstamp	Time of message	Seconds after midnight	0-86399	INT32
julian_date	Departure date	Number of days since 1/1/1980		short
etd	Estimated departure time	Minutes after midnight	0-1439	short
eta	Estimated arrival time	Minutes after midnight	0-1439	short
td	Departure time	Minutes after midnight	0-1439	short
ta	Arrival time	Minutes after midnight	0-1439	short
flight_id	Aircraft ASCII indentifier	1 or 3 letters and up to 4 numbers		string7
dept_aprt	Airport of origin	3 to 5 letters & numbers		string5
arr_aprt	Destination airport	3 to 5 letters & numbers		string5
acft_type	Type of aircraft	Land, copter, etc	unkn, land, heli, amph, sea	string4
dept_arr_flgs	Packed waypoint flag & physi-cal, user, and weight class	See below		char
srce	Source NAS message type	See below	D,L,E,B,H	char
classes	Packed waypoint flag & physi-cal, user, and weight class	See below		char
num _aircraft	Number of aircraft in formation	From field 3		char
ac_eqp_prefix	Aircraft equipment prefix	From field 3		char
ac_eqp_suffix	Aircraft equipment suffix	See below		char
ac_rmk_bitflags	Flags for field 11 keywords	16 bitflags	0.1	char
geographical_flags	Flags for future use	unused		char
pad	Space holder to keep structure an even size			char

The **srce** field can have one of the following values.

D=DZ

L=AZ

E=EDCT

B=5 SETBACK

H=CONTROL CANCEL

Table 25-21. TTM-FTM TZ Transaction Data Structure

TTM-FTM TZ Transaction				
Library Name: fdb_openlib		**Purpose:** Hold TZ data for transfer from FDB to client FTMs		
Element Name: fdbftm_pack.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
flight_indx	Internal identifer.			INT32
tstamp	Time of message	Seconds after midnight	0-86399	INT32
ctrt_spd_alt	Packed center of origin of message, speed, & altitude	See below		INT32
julian_date	Departure date	Number of days since 1/1/1980		short
lat	Last reported latitude	Minutes of arc	-32768 –5400	short
lon	Last reported longitude	Minutes of arc	-32768 –5400	short
nxt_lat	Latitude of the next event	Minutes of arc	-32768 –5400	short
nxt_lon	Longitude of the next event	Minutes of arc	-32768 –5400	short
eta	Estimated arrival time	Minutes after midnight	0-1439	short
etime	Last message time	Minutes after midnight	0-1439	short
flight_id	Aircraft identifier	1 or 3 letters and up to 4 numbers		string7
wypt_flag	Ghosting flag	0=Ghost to waypoint	0-1	char/Bit 7 (Bits6…0 Unused)
altitude_type	Either transitioning or cruising mode		C,T	char
arr_fix_time	Estimated time at arrival fix	minutes after midnight	0-1439	short
geographical_flags	Flags for suture use	unused	unused	short
pad	Space holder to keep structure an even size			char

Table 25-22. fsatat_spd_alt Data Substructure

fatat_spd_alt				
Library Name: fdb_openlib		**Purpose:** Packed flight status, speed, and altitude		
Element Name: fdbftm_pack.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
unused				31 – 28
flight_status	Current state of flight	See below	N,S,F,A,R,C,D,T, X,E	27 24
unused				23 – 22
altitude	Flight's altitude	Hundreds of feet		21 –12
speed	Flight's speed	Nautical miles per hour		11 - 0

The **flight_status** field can have one of the following values:

N = None

S = Scheduled

F = Filed

A = Active

R = Ascending

C = Cruising

D = Descending

T = Completed

X = Cancelled

E = Error

Table 25-23. sizes Data Substructure

sizes				
Library Name: fdb_openlib		**Purpose:** Packed number of waypoints ,sectors,fixes, airways, & centers		
Element Name: fdbftm_pack.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
unused				31 – 29
waypoints	Number of characters in waypoints list		0 – 70	28 – 22
sectors	Number of characters in sectors list		0 – 50	21 –16
fixes	Number of characters in fixes list		0 – 50	15 – 10
airways	Number of characters in airways list		0 – 50	9 – 4
centers	Number of characters in centers list		0 – 10	3 - 0

Table 25-24. classes Data Substructure

classes				
Library Name: fdb_openlib		**Purpose:** Packed waypoint flag & physical, user, and weight class		
Element Name: fdbftm_pack.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
waypoint	Ghosting flag	0 = Ghost to waypoint	0 – 1	7
physical_class	Aircraft physical class	See below	1 – 4	6 – 5
user_class	Aircraft user class	See below	0 – 5	4 – 2
weight_class	Aircraft weight class	Small, Large, Heavy	1 – 3	1 - 0

The **physical_class** field can have one of the following values:

1=Piston

2=Turbo

3=Jet

4=Helicopter

The **user_class** field can have one of the following values:

0=Other

1=Air Taxi

2=Cargo

3=Commercial

4=General Aviation

5=Military

Table 25-25. dept_arr_flgs Data Substructure

dept_arr_figs				
Library Name: fdb_openlib		**Purpose:** Packed flags denoting whether time is estimated or actual		
Element Name: fdbftm_pack.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
departure_flag	Specifies derivation or departure time	See below	1 – 5	7 – 4
arrival_flag	Specifies derivation of arrival time	See below	1 – 5	3 - 0

The flag values must be one of the following:

1=actual

2=estimated

3=controlled

4=proposed

5=scheduled

Table 25-26. spd_alt Data Substructure

spd_alt				
Library Name: fdb_openlib		**Purpose:** Packed speed and altitude		
Element Name: fdbftm_pack.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
unused				31 – 22
altitude	Flight altitude	Hundreds of feet		21 – 12
speed	Flight speed	Nautical miles per hour		11 - 0

Table 25-27. ctr_spd_alt Data Substructure

spd_alt				
Library Name: fdb_openlib **Element Name:** fdbftm_pack.h		**Purpose:** Packed center of origin, speed and altitude		
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
center	Center of origin of message	See ARTCC center codes		31 - 24
unused				23 – 22
altitude	Flight altitude	Hundreds of feet		21 – 12
speed	Flight speed	Nautical miles per hour		11 - 0